Defence Research and Development Canada
Recherche et développement pour la défense Canada

Internet Based Robot Control Using CORBA Based Communications

A study into the simplification of multirobot control

S. Verret and J. Collier
DRDC Suffield

A. von Bertoldi
ADGA Inc.

Defence R&D Canada

Technical Memorandum

DRDC Suffield TM 2009-127

December 2009

Canada

Internet Based Robot Control Using CORBA Based Communications

A study into the simplification of multirobot control

S. Verret and J. Collier
Defence R&D Canada – Suffield

A. von Bertoldi
ADGA Inc.
125 Taravista Way NE
Calgary, Alberta
T3J 4K8

Defence R&D Canada – Suffield

Technical Memorandum

DRDC Suffield TM 2009-127

December 2009

Principal Author

Original signed by S. Verret

S. Verret

Approved by

Original signed by D.M. Hanna

D.M. Hanna

Head/AISS

Approved for release by

Original signed by Dr P.A. D'Agostino

Dr P.A. D'Agostino

Head/Document Review Panel

Abstract

Researchers in the field of robotics have been seeking methods to both control and monitor their vehicles. Unfortunately the programs they have developed to perform these tasks are normally dependent on the robotic software infrastructure or are very difficult to understand for an outside user. This paper looks to tackle the problem of monitoring and controlling a robotics system using a web browser. The goal of this paper is to describe the potential for a system that will control and monitor a CORBA based robotics framework from a simple HTTP based browser.

Résumé

Les chercheurs dans le domaine de la robotique ont cherché des méthodes qui à la fois commandent et surveillent leurs véhicules. Les programmes qu’ils ont développé pour effectuer ces tâches sont malheureusement dépendants de l’infrastructure des logiciels en robotique et sont généralement difficiles à comprendre pour un utilisateur extérieur. Cet article cherche à s’attaquer au problème de la surveillance et de la commande d’un système de robotique qui utilise un navigateur Web. Le but de cet article est de décrire le potentiel d’un système capable de commander et surveiller un cadriciel en robotique basé sur CORBA à partir d’un simple cadriciel basé sur HTTP.

This page intentionally left blank.

Executive summary

Internet Based Robot Control Using CORBA Based Communications

S. Verret, J. Collier, A. von Bertoldi; DRDC Suffield TM 2009-127; Defence R&D Canada – Suffield; December 2009.

Background: Scientists at Defence R&D Canada – Suffield investigate software architectures for unmanned systems. The Autonomous Land Systems (ALS) and Cohort projects, undertaken by the Autonomous Intelligent Systems Section (AISS), resulted in the Architecture for Autonomy (AFA), a distributed software framework and design methodology for unmanned systems. Despite the large effort in developing the AFA, no effort has been undertaken to develop a graphical user interface (GUI) to interact with the AFA for multirobot high level control.

This paper investigates various design solutions for a web browser based GUI to perform high level control of heterogeneous multirobot systems. A browser based solution is preferred as it is operating system independent, can be distributed across multiple robots, harnesses the powerful capabilities of today's browsers and web-based toolkits, and greatly simplifies/eliminates GUI configuration and installation.

Principle Results: Numerous software packages, toolkits and scripting languages were evaluated for their applicability to web based robot control. Criteria for selection included ease of deployment, maintainability, scalability, support for existing infrastructure, etc. Additionally, software design methodologies were evaluated including different browser, and server side configurations. This rigorous evaluation resulted in a preliminary design solution.

Significance of Results: Adoption of the tools and design methodology will result in a GUI that is responsive, scalable, easy to deploy, operating system independent, and easy to develop. The proposed solution cleanly separates the GUI from the application (i.e., the AFA). For these reasons, the GUI will be easily maintained and adapted for future research.

Future Work: Development of the proposed GUI will be incremental based on the design decisions presented in this paper. The modularity of the AFA, and thus, the web based GUI enables a simple interface to one robotic element (e.g. Odometry) to be designed, tested, and evaluated to assess the performance of the methodology. Refinements and/or design changes may occur as a result of this testing. The software architecture will solidify as more modules are developed. Aesthetic design and display

issues can be addressed as needed and should not affect the actual design methodology and toolset. The proposed GUI is expected to be used for the AISS research in the forseeable future with continued development as new autonomous capabilities are integrated into the AFA.

Sommaire

Internet Based Robot Control Using CORBA Based Communications

S. Verret, J. Collier, A. von Bertoldi; DRDC Suffield TM 2009-127; R & D pour la défense Canada – Suffield; Décembre 2009.

Contexte : Les scientifiques de R & D pour la défense Canada – Suffield examinent les architectures de logiciels pour les systèmes sans équipage. Les projets des Systèmes terrestres autonomes (STA) et Cohort, entrepris par la Section des Systèmes intelligents autonomes (SSIA) ont résulté en l'Architecture d'autonomie (Architecture for Autonomy), un cadriciel de logiciels distribué et une méthodologie de conception pour les systèmes sans équipage. En dépit de l'effort important qui a consisté à développer l'Architecture d'autonomie, aucun effort n'a été entrepris pour développer l'interface graphique (GUI) qui interagit avec l'Architecture d'autonomie pour la commande multirobot de haut niveau.

Cet article examine les solutions de concepts variés pour un GUI basé sur un navigateur Web qui vise à effectuer des commandes de haut niveau de systèmes multirobot hétérogènes. On préfère une solution basée sur un navigateur parce qu'il s'agit d'un système d'exploitation indépendant qui peut être distribué à travers des robots multiples, qui peut aménager les capacités puissantes des navigateurs d'aujourd'hui et des boîtes à outils logiciels et qui simplifie et élimine grandement la configuration et l'installation GUI.

Résultats principaux : De nombreux progiciels, boîtes à outils logiciels et langages de script ont été évalué en vertu de leur applicabilité à la commande de robot basée sur le web. Les critères de sélection incluent la facilité de déploiement, de maintenance, de variabilité d'échelle, de soutien de l'infrastructure existante, etc. De plus, les méthodologies de concept de logiciels qui ont été évaluées comprennent des configurations différentes de navigateurs et de côtés serveurs. Cette évaluation rigoureuse a résulté en une solution de concept préliminaire.

Portée des résultats : L'adoption d'une méthodologie de concepts et d'outils résultera en un GUI qui est réceptif, échelonnable, facile à déployer, un système opérationnel indépendant et facile à développer. La solution proposée sépare nettement le GUI de l'application (ex.: Architecture d'autonomie). C'est pour cette raison que le GUI sera facilement maintenu et adapté à la recherche future.

Perspectives d'avenir : Le développement du GUI proposé sera incrémentiel et basé sur les décisions de concepts présentées dans cet article. La modularité de l'Architecture d'autonomie et par conséquent du GUI basé sur le Web permet à la simple interface d'un élément robotique (ex. : odométrie) d'être conçue, testée et évaluée pour connaître le rendement de la méthodologie. Les raffinements et les changements de concepts pourront résulter de ce test. L'architecture de logiciel se solidifiera alors que des modèles supplémentaires seront mis au point. Les problèmes d'esthétique des concepts et d'affichage pourront être abordés selon les besoins et ne devraient pas affecter la méthodologie du concept et de la boîte à outils logiciels actuels. On prévoit que le GUI proposé sera utilisé pour la recherche des SSIA dans un avenir prévisible alors que de nouvelles capacités sont intégrées dans l'Architecture d'autonomie avec le développement continu.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

The Autonomous Land Systems (ALS) and Cohort projects, undertaken by the Autonomous Intelligent System Section (AISS) at Defence R&D Canada – Suffield, research and develop autonomous unmanned collaborative robotic vehicles. After a thorough review of existing robotics toolkits, the Miro framework was chosen as a basis on which to build the Architecture for Autonomy [1]. The Miro framework uses the TAO implementation of the CORBA specification that relies on the ACE toolkit for interprocess communication (IPC). TAO is a real-time implementation of the CORBA specification for object-based communication and remote method invocation (RMI). CORBA is programming language and operating system independent and network transparent. These features make it an extremely powerful and flexible architecture for distributed robot communication and cooperation.

The ALS software base includes ACE, TAO, Miro, DRDC's additions to Miro (called drdcMiro) and a large number of other libraries that provide underlying functionality to these modules. As a result, the ALS software base is large and complex, and requires considerable time and effort to install and learn. Additionally, to interact with any of the TAO, Miro or drdcMiro robot service objects one is required not only to have a functioning software environment (properly installed and configured), but also to develop software components including a graphical user interface to perform the interaction. This is a time consuming and non-trivial task, and necessary for even simple interactions. This situation is inconvenient and limiting. It is highly desirable to enable a method to interact with existing TAO, Miro and drdcMiro components that does not rely on this large and complex software environment, is simple and convenient to use, and is consistent for all of the available Miro/drdcMiro services/objects. Like CORBA itself, this method should be programming language and operating system independent. Furthermore, it should be easy for those with no knowledge of CORBA or Miro to learn and interact with, and thus should hide all of those details from the user. Finally, it should also provide an interface or API for developers that wish to access Miro services through this mechanism instead of the traditional CORBA-based route.

The World Wide Web (WWW) and web browsers are an ideal platform to enable this aim. The WWW is an open, operating-system and language independent platform accessible from anywhere in the world. Furthermore, web-browsers are an ideal client as they are capable of producing rich interfaces, are comparatively easy to program, and are installed in nearly every desktop computer.

The goal of this work is to develop a method by which standard web browsers can interact with and access the CORBA-based Miro and drdcMiro service objects, on a robot and thus enable web-based monitoring and control of the robot. This implies a constraint on possible solutions: Web browsers generally only support the HTTP and

FTP communication protocols[1].Any communication between the client (i.e. the web browser) and the robot computer (i.e. the server) must use one of these protocols. The FTP protocol is not appropriate for this goal, thus the HTTP must be the communication protocol between the client and server. An additional reason for this constraint is that there can be any number of internet devices such as firewalls, gateways and routers between the client and server, and it cannot be assumed that these devices will allow traffic from protocols other than HTTP.

The remainder of this document is presented as follows: Section 2 gives a brief overview of the current CORBA based architecture, describing how the software components interact and communicate with each other and highlighting its main advantages and disadvantages. This leads to a motivation for this work, and a detailed description of the problem. A set of criteria by which to evaluate the alternatives is presented. Finally, the four main approaches to solving this problem highlighted from the literature review are described. Section 3 introduces the relevant technologies, including specific tools and architectures that could contribute, wholly or in part, to a solution. Each piece of technology and its potential role in a solution is described, and examples of existing uses of the technology in the problem domain are presented. Section 4 presents possible solutions in terms of combinations of specific technologies and architectures. Section 5 discusses the advantages, disadvantages and implications of the potential solutions. Finally, Section 6 describes the selected and the details of its implementation.

2 Communications

2.1 Current System

As mentioned in Section 1, the Miro robotics middleware was chosen by the AISS group as the foundation on which to develop their distributed autonomous systems capabilities, and includes the components described below. All of these software components are available and can be used to implement a solution to the current problem.

CORBA
CORBA, an interprocess communication and remote method invocation specification, allows clients to invoke operations on distributed objects without concern for object location, programming language, operating system, communication protocols and interconnects or hardware. All robot services in AISS's framework are created as CORBA objects. Any web-based clients wishing to interact with the robot services will at some point have to do so via CORBA standards.

[1]some browsers also support the bittorent and other lesses known protocols though plugins, but for compatibility reasons only HTTP and FTP can be assumed.

CORBA Interface Definition Language (IDL)

CORBA allows software processes written in different languages to interact through the Interface Definition language (IDL). IDL is a language used to define the interface to a particular CORBA object. An IDL compiler produces client and server-side software stubs for a particular language that allows clients to request data from and invoke methods in the server. For any CORBA service to be accessible to clients, an IDL must exist for the service, and the client and server stubs generated by the IDL compiler must be included in the client and server implementation respectively.

CORBA Dynamic Invocation Interface (DII)

DII, in contrast to IDL, is a method that allows clients to interact with services without requiring the IDL stub for that service. The DII allows a client to identify operations, parameters and types in an object via string names. DII is more complex and has poorer performance than IDL, but is more flexible. Any web-based clients wishing to interact with the robot services will have to use IDL or DII.

CORBA NamingService

The NamingService is a process in which CORBA service objects register their service. Clients can query the NamingService for a particular service and obtain a reference to the service, which they can subsequently use to interact with it. Any web-based clients wishing to interact with the robot services will at some point have to interact with the NamingService.

ACE

ACE (Adaptive Communication Environment) is a powerful, operating system-independent toolkit for concurrent communication between software components that takes care of many of the details of interprocess communication, including event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization [2]. Although it can be used to interact with the robot services, it is not a necessity.

TAO

TAO is an implementation of the real time CORBA specification built upon the ACE toolkit [3]. The TAO package includes an IDL-to-C++ compiler and a NamingService. All CORBA operations are done through TAO.

Miro

Miro is a distributed framework for robot control built upon ACE and TAO, and thus inherits all of the features of ACE/TAO. Miro provides many common robot services such as odometry, motion, range sensors, video and many more, all developed as CORBA services using TAO.

2.1.1 Implications

CORBA is very powerful, object-oriented and network-transparent method of IPC and RMI. It allows a developers to compose robots as a collection if individual CORBA objects that can communicate and interoperate. Furthermore, some robot-related processes are computationally intensive; this framework allows such services to be run on a separate, dedicated computer in a transparent way.

While the CORBA specification is language agnostic, specific implementations of the specification use a particular language; in TAO's case C++. To interact with a CORBA object it it generally wise (for reasons of compatibility and code re-use) to implement clients using the facilities provided the CORBA implementation, such as it's IDL compiler. As a result, a developer is, in practice, tied to the particular implementation of CORBA and it's programming language. This fact is not inherently a problem, other than it results in a large and complex software dependency; in this case ACE, TAO, Miro and drdcMiro.

2.2 Problem Definition

The goal of this work is to develop a method by which standard, unmodified[2] web browsers can interact with and access the Miro service objects on a robot, and thus enable web-based monitoring and control of the robot. Furthermore, clients must not have any dependency whatsoever on the ACE, TAO, Miro or drdcMiro packages.

2.2.1 Constraints

Implicit in this goal are a number of constraints that will shape the solution: The current variety of web browsers support a number of protocols including HTTP, FTP, POP3, SMTP, Bittorrent, etc. No current browsers support the IIOP or GIOP protocols which CORBA uses to communicate. Many browsers allow the addition of protocols though configuration, e.g. a URL for the new protocol is associated with a program on the computer. Whenever the URL for the new protocol is entered into the address bar of the browser, it will call the external program to handle the protocol requests and responses. The goal of this project could be realized using this feature by developing an application that would communicate directly with the CORBA object via IIOP when a new URL was entered. However, this would have the client depend on the entire AISS software base, which is explicitly against the goals of this project. This approach is therefore not feasible. Consequently, the solution must rely on one of the browser's built-in protocols. Furthermore, of the built-in protocols, only HTTP is appropriate as it is the only one that allows multi-media content and can be safely assumed not to be blocked by firewalls. The solution must therefore

[2]by unmodified we mean no source-code modifications; configuration changes are acceptable.

rely on the HTTP protocol, which in turn implies the use of an HTTP server in the solution.

2.2.2 Challenges

In view of the constraints implicit in this project's goal, there are a number of core challenges to overcome in order to develop a solution to web-based robot control.

HTTP-IIOP Gateway
Web browsers do not understand IIOP requests and, likewise, CORBA objects do not understand HTTP requests. This requires converting requests made by the client over HTTP to the appropriate IIOP request from a CORBA object, and converting the response from the CORBA service into a representation that can be digested by the client, i.e. an HTML document. This HTTP to IIOP gateway will likely be the largest single component of the final solution. This gateway will, on one side, service HTTP requests made by the web client and on the other, act as a client of the CORBA service. This component therefore requires the ability to communicate with the HTTP server and CORBA objects. Communication with CORBA objects will be enabled via IDL or DII, while communication with the HTTP server could occur using a number of methods. Finally, this component of the solution will depend in the AISS software base and should therefore reside on the same computer as CORBA services themselves.

Client UI
Each autonomous entity will be composed of a number of CORBA objects, each providing one or several services. These services will be varied: some will provide information about a robot component, while others will allow certain parts of the robot to be affected or controlled. All of these services should be accessible and controllable from the client. It is necessary to create a client interface that not only provides access to all the functionality of the underlying individual CORBA service objects, but groups and presents them in a manner that results in an informal, intuitive and highly usable interface to the autonomous entity as a whole.

Resource Addressing
A method must exist for the web browser client to refer to a specific CORBA object residing on a robot, and possibly a specific method on that object. A scheme must be developed to associate a web browser addressable URL to each of the CORBA services on a robot, and possibly each of the methods defined in that object's interface.

Message Format
Client/server communication must take place over HTTP. However, as discussed in Sections 4 and 6, limiting the client-server transaction to HTTP requests and HTML responses has several disadvantages. Therefore, a message format to represent client

request and service responses must be developed. This message format must be able to represent all relevant CORBA interface data including: CORBA namespace, the CORBA object name, the object's desired method name, the method parameter names and types, its return parameter names and types and any error messages. This message format should be standardized to facilitate the creation of other types clients that can consume the provided service.

Figure 1 summarizes the various communication components that are the current basis of AISS's communication infrastructure, and the components that will form part of the internet-based robot control solution. The existing CORBA clients server objects are on the bottom. Servers and clients must employ the IDL skeleton and stub respectively, which are created by compiling the IDL service definition. Servers register their service with the Naming Service and clients query the Naming Service to get a reference to the service object. Clients invoke the desired method on the server via the IDL stub, and receive a response from the server when it has completed servicing the request. The TAO implementation of the CORBA specification provides all CORBA functionality including the Naming Service, object request broker, object adapter and IDL compiler. Miro robot services are implemented as CORBA server objects, and can be accessed by any CORBA client. The HTTP/HTML based world wide web and WWW browser client are on top. Web browsers, as WWW clients can only communicate over HTTP and display information as HTML documents and multimedia. Shown in the middle is the component - to be developed by this project - that will enable the HTTP based WWW communication network interoperate with the IIOP/GIOP-based CORBA communication network. This component must act as a WWW server on one side, and a CORBA client on the other, meaning it must understand both the HTTP and IIOP/GIOP protocols. A web browser-based client user interface to the Miro services must also be be developed.

2.3 Criteria

The following criteria were considered when evaluating and comparing technical components such as specific technologies and architectures that may be used in the solution:

2.3.1 Existing Infrastructure

The solution should take advantage of the existing infrastructure as much as possible; this applies **only** to the service component. This includes using components of the TAO, Miro and drdcMiro modules and the C++ language whenever possible. The reasons for this are three-fold: Using the Miro and drdcMiro modules whenever possible will prevent the duplication of functionality, which in turn improves maintainability. It will also allow developers of the web-based control solution to draw on

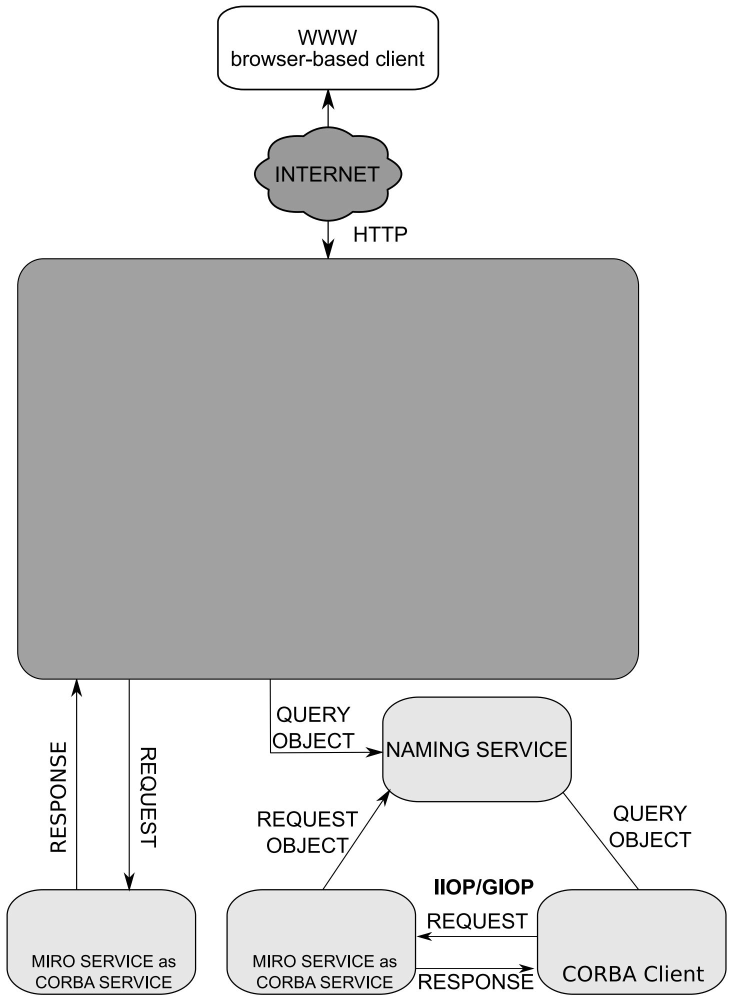

Figure 1: *Components of the communication framework. Shown are AISS's existing CORBA-based communication architecture (bottom), HTTP-based world wide web and web browser client (top), and the interoperability component (webMiro) to be developed in this work (middle).*

AISS's knowledge base with these modules. Finally, reusing these modules will lower the entry barrier for learning this system. This consideration must be reconciled with choosing the most appropriate tools for a particular task, and with performance considerations.

2.3.2 Authentication

It is unclear at the time of writing of this document if user authentication and security will be requirements of the solution. This must be considered when selecting technologies and architectures in the event that user authentication and security become requirements.

2.3.3 Multiple Clients

It has not been specified at the time of writing of this document how multiple clients should be handled. That is, how should requests to the same CORBA service from different clients be handled? This will be most important when, for example, two clients request a service which changes the robot state such as position control. While this policy can be addressed at a later date, the selected technologies should have the capability to receive, distinguish between, and service multiple clients. This capability includes the ability to efficiently handle multiple requests, such as multi-threading or multiplexing of requests.

2.3.4 Performance/Server Load

The web-based control software processes should not interfere with or excessively affect normal operation of the robot services. As such, they should be efficient in their use of CPU and memory resources.

2.3.5 Communication Delay

Communication delay between the client and server is a potential performance bottleneck from the client's point of view. Any solution must address this by minimizing the execution time of all components, minimizing communication-based delays and overhead costs (such as constantly opening sockets), and minimizing the amount of data being transferred between client and server.

2.3.6 Media Types

It has not been specified at the time of writing of this document the types of media that may be used to present information to the user in the client. In addition to text, other media such as audio, video and animations could be used to display the status of the robot services. For example, data from the SonarPoll service, a service that polls for data from a sonar sensor, might by displayed more effectively as an animation as opposed to text. Similarly, it would be desirable to display images generated by services such as camera services. The technologies must also be evaluated by their ability to produce or display different media types.

2.3.7 Separation of Logic and Presentation

It is generally considered good practice in web application development to separate the presentation layer (i.e. HTML document generation) from the application logic (interaction with the CORBA objects/services). This allows the client interface to be developed and modified independently of the application logic, and makes it possible to provide multiple interfaces to the same service. For example, if control from web-enabled portable devices such as PDAs or phones is desired, the client interface developed for computers may not be adequate. A different interface could then be developed for mobile devices without the need for changes in the application logic. The proposed tools and architectures should be evaluated against their adherence of this practice.

2.3.8 CORBA Support

Compatible CORBA support is considered advantageous for this problem.

2.3.9 Ease of Deployment

Solutions and technologies were evaluated against their development and maintenance burden. This includes the complexity of configuration required to install and maintain a particular technology, and the complexity of installing and maintaining the number of components in a solution.

2.3.10 Maintainability

The AISS software stack is already large and complex. The addition of the webMiro module should not add an undue level of additional maintenance effort. Maintainability means minimizing the effort required to implement changes both directly associated with the webMiro module, such as bug fixes and features additions, and changes required to this module in response to changes to existing or addition of new Miro and drdcMiro services.

2.3.11 Scalability

The chosen solution should scale well in terms of maintenance and performance in response to addition of new services.

2.3.12 Open Source

Where the feature-set and performance is comparable, current and active open source projects are given preference over proprietary alternatives when selecting software components to develop a solution for this project.

2.4 Existing Work

Kumar et al. [4], published a conference proceedings paper titled, "Accessing CORBA Objects on the Web" and highlighted four main techniques to allow web browser-based clients to access remote CORBA objects:

- **Browser embedded Java applets:** This approach involves developing a Java applet that communicates directly with the CORBA object over IIOP or some other custom protocol, bypassing the HTTP server altogether.
- **Extensions to web browser to service the IIOP protocol:** This approach involves extending the capabilities of web browsers to handle the IIOP protocol and communicate with CORBA objects directly.
- **Custom HTTP-IIOP gateway/proxy:** This approach involves configuring web browsers to connect to a custom proxy which acts as a gateway between the HTTP and IIOP protocols.
- **Web services approach:** This approach involves using a web-services stack to present the CORBA service as a web service, so that it may be accessed by web clients.

Each approach attempts to solve the problems mentioned in Section 2.2, while some eliminate them completely. Examples and a brief discussion of each of these approaches is given in Section 3 along with the presentation of the corresponding technology.

3 Internet Based Control

This section introduces the various technologies that could form part of the solution to web-based robot control. They are introduced in roughly the same order as outlined in section 2.4, grouped with related technologies where applicable. Section 3.1

presents technologies specific to WWW clients that could form part of a solution, including approaches 1 and 2 in 2.4. Section 3.2 corresponds to approach 3 from 2.4. Sections 3.3, 3.4 and 3.5 present a plethora of technologies belonging to approach 4: Web Services. All of these technologies aim to provide the same functionality, which is to allow invocation of applications on a server computer from a WWW-based client on another computer, and return the result of the operation back to the client in the form of an HTML document. Section 3.6 will present a number of architectures for implementing Web Services. Finally, section 3.7 will present existing tools to facilitate development of Web Services servers and clients.

3.1 Client-Side Technologies

Client side technologies operate within the web browser to generate dynamic content. This approach, offloading processing onto the client, has several advantages over approaches that interact with a server for all operations. Because communication with the server is reduced, and thus its associated delay, these UIs are much more responsive than the alternative, and provide a much better user experience. This also results in reduced demand on the server, which allows it to service more clients. Java Applets, JavaScript and Flash all fall into this category and are aimed at enabling rich, dynamic client interfaces with all processing and presentation generation on the client side.

3.1.1 Java Applets

Some of the early examples of robot control over the web using a standard web browser relied on Java Applets. A Java applet is a small application compiled to Java-Byte-Code that is downloaded by the web browser and run within the browser by a Java Virtual Machine (JVM) [5]. Applets can also be run outside of the web browser by the JVM. An applet can be programmed using many of the same tools and libraries as a normal Java application, including TCP/IP and even CORBA libraries.

Applets are typically composed of widgets such as buttons, input fields, drawing areas, etc., much like GUI applications, except that the widgets interact with processes running in a remote location. The main goal of applets were to provide a more rich, dynamic and responsive web client compared to traditional HTML web pages by moving much of the processing to the client, thus reducing client-server interaction and its associated delays.

Web-based robot control using applets is accomplished by creating one applet for each robot service, e.g. odometry, velocity control, etc. The applet would contain all the relevant widget elements required to fully represent and interact with the corresponding CORBA service object. A web page can contain a single applet corresponding to one service, or several applets each corresponding to a different/unique service running either in the same or different computer. The applet connects directly to the robot service, typically via TCP/IP, bypassing the HTTP/Web server all-together. The browser thus becomes nothing more than a platform on which to run the applet. Java includes CORBA bindings, so it is possible for applets to interact directly with a CORBA objects/services over IIOP. This approach has been used by [6, 7, 8, 9] to achieve web-based robot control.

Applets are an initially attractive solution as it is relatively simple to implement, and as communication with server over TCP/IP is relatively low bandwidth should provide good performance when interacting with the server. Furthermore, this method requires nothing on the server side other than a standard HTTP server to serve the initial page that contains the applet(s) embedded within it. However, applets suffer from a few significant drawbacks: Direct connections to the server process via TCP/IP or IIOP can be blocked by firewalls which typically only allow HTTP traffic for security. Not all browsers are compatible with and able to run Java Applets. These issues will require some limitations on the final solution if applets are to be part of it, specifically, guaranteed access over the desired protocol and the restriction to supported browsers. Finally, the popularity of applets has decreased recently with the proliferation of Ajax and Flash technologies.

3.1.2 Browser Extension

Yet another method to implement web-based CORBA object/service browsing is through extending a web browser to include support for the IIOP protocol. Kumar et al. propose this approach in [4]. They propose a URL scheme to address CORBA service objects as follows:

```
iiopname://objectServer.com/objectName::methodName?par1=val1&par2=val2
```

where *objectServer* is the host name or IP address of the computer on which the CORBA services reside, *objectName* is the name of the CORBA object as registered in the NamingService, *methodName* is the name of the object method we wish to invoke and *parX* and *valX* are the names and values of the method parameters. The authors also present an attractive method to navigate a computer hosting CORBA objects/services. If the user enters only the host name, the browser should query the NamingService and return a web page listing each of the available CORBA services and corresponding URLs available in that computer. When the user navigates to one of the services, the browser should return a web page listing all methods provided by that CORBA service, including parameter names and types and corresponding URLs.

The authors implement this by extending a HotJava browser to support the new protocol, and must also develop a proxy server as all the required functionality could not be added to the HotJava browser. Unfortunately, they do not describe any of the implementation details, or if the approach was successful.

As with Java Applets, this approach would have the browser communicate directly with the CORBA objects/services. Also as with applets, this requires no additional components on the server side. This approach also shares some of the disadvantages of applets, namely that communication is over IIOP on a non-standard port that could be blocked by firewalls. It is not a suitable solution as it requires considerable modification to the web browser, certainly more than could be accomplished by a FireFox extension for example. Despite not being a viable solution, Kumar et al. present some interesting and attractive ideas that could be incorporated into other solutions. For instance, the object addressing and browsing scheme is a powerful and intuitive approach to discovering, browsing and interacting with CORBA objects over a web browser.

3.1.3 Ajax

Ajax is a collection of web technologies aimed at providing a more dynamic, interactive and responsive web browsing experience. At the core of Ajax are the JavaScript client-side scripting language and the XMLHTTPRequest object which can make asynchronous requests to the server. JavaScript can access the browser's Document Object Model (the browsers internal representation of the web page) allowing it to dynamically update the content of the web page with data returned from the XMLHTTPRequest without having to reload the entire web page. Because the XMLHTTPRequest method is asynchronous, the web page is still responsive and can be manipulated by the user while the request is being processed. Additionally, because the data transmitted and rendered in an XMLHTTPRequest is much smaller than a typical HTTP GET or POST call, which could return a complete web page, and because only a small portion of the web page is updated, the response time of a web-page that includes Ajax will be much improved. The data sent and received in the XMLHTTPRequest call can be formatted as XML, JSON, SOAP envelopes, HTML or plain text, each of which has advantages depending on the application. Note that unlike Java Applets and browser extensions described above, Ajax alone cannot enable web control though it can be a part of the solution.

There are currently no published works detailing the use of Ajax in web-based robot control, only public accounts that give very brief descriptions and implementation details [10, 11].

The main advantage Ajax provides is decreased delay in completing a server request and increased web page responsiveness. Additionally, the ability to connect an XMLHTTPRequest to a specific action on the server and subsequently to a callback function in the client which is automatically triggered when the server request is complete, results in a system that can be programmed and behaves more like a typical, event-driven GUI interface than a web page.

Ajax's main disadvantages are compatibility across web browsers brought about by varying implementations of JavaScript and XMLHTTPRequest, and of course its reliance on JavaScript. Well established techniques exist for dealing with both however; detecting the browser and providing browser-specific code for the former and providing static HTML equivalents if JavaScript is disabled for the latter.

Finally, several toolkits exist to facilitate developing Ajax web applications [12, 13, 14, 15, 16, 17], though some of these rely on specific server side components. These toolkits provide libraries of UI widgets and generally automatically handle browser incompatibility issues.

Comet
Comet, also referred to as Ajax-push or server-push, is an Ajax-related concept by which a server *pushes* data to interested clients asynchronously without requiring the client to request it. It allows data created on a server to be pushed to the client immediately, without the client having to poll the server continuously for new data.

Comet is not a technology however, only a concept, and no standard implementation of Comet exists. Several implementations of the Comet model exist, each with their own advantages and disadvantages. These advantages and disadvantages revolve around the method used for circumventing the 2-open-connections-per-client-per-domain limit specified in HTML/1.1 which is not necessarily followed by most browsers and most of which require a specific server-side component to service the requests.

Although no evidence was found of Comet being used in web-based robots, it is used widely in web services and web applications and could be a useful tool, especially for services that use the publish-subscribe pattern information.

3.1.4 Adobe Flash

Like Java Applets and Ajax, Flash is another platform for creating so called rich internet applications, and includes development tools and a player client both standalone and browser embeddable. The term Flash is more commonly used to refer to a Flash applications, authored with CS3[3]or other tools, and run by the Flash player. Flash supports Scalable Vector Graphics (SVG), streaming video and audio, and has a scripting language similar to JavaScript called ActionScript. Flash is used extensively for delivering multimedia rich web content such as video and animations, and more recently for interactive web applications. Depending on the applications being

[3]Adobe's flagship Flash IDE

developed, Flash applications require much less programming to develop. A simple application can be developed with no programming at all compared to the equivalent Ajax or Java Applet web application. Flash applications are saved as Shockwave Flash (SWF) files, embedded in an HTML file and can be served by HTTP web servers. Serving streaming audio and video requires a Real Time Messaging Protocol (RTMP) server.

As a solution, Flash sits somewhere between Ajax and Java Applets. It is like applets in that it is a compiled application embedded in the web page and requires a runtime engine to run. It is like Ajax in that Flash can only be part of solution; specifically the client interface as Flash applications lack the capability to connect directly to CORBA objects on the server. Flash is better suited than Ajax for multimedia content such as streaming audio/video, though client interaction is limited to the Flash application's frame. Ajax is much better suited for text manipulation and interaction with the entire web page through its ability to directly manipulate the DOM.

Flash was used to create the interface in [18] to control a robotic arm over the internet. Their solution included an HTTP server that served the interface web page with an embedded Flash application, and ran a PHP script in response to GET/POST requests. The PHP application connected to a standalone process through TCP/IP sockets that directly controlled the robot.

Flash playback is reasonably well supported across browsers and operating systems. There are however very few official tools for authoring Flash applications on Linux [19]. A number of open source authoring tools are available [20, 21], though none have the full range of features available with Adobe products. Open source RTMP servers capable of serving streaming video are also few [22]. Although Flash is proprietary technology, many of its specifications have been published and are now public.

3.2 Custom HTTP-IIOP Proxy

A project commissioned by the ANSA [23] group undertook the task of creating a universal solution to allow interoperability between CORBA and the WWW. It was a large, long-term project with support from commercial partners. The project's goal was to make CORBA services available to WWW clients and vice versa, and to remain compatible with existing WWW and CORBA practices, specifications and technologies.

Among several proposed solutions was the creation of HTTP/IIOP Proxies [24]. This solution included three components: an HTTP-to-IIOP proxy labeled H2I, an IIOP-to-HTTP proxy labeled I2H and a locator entity. WWW clients requiring access to CORBA object would be configured to connect to an H2I proxy. The H2I proxy would be responsible for intercepting HTTP requests made by an WWW client, converting them to the equivalent IIOP requests for a CORBA object, and converting the reply from the CORBA object back into a response suitable for consumption by a WWW client. Similarly, the I2H proxy was responsible for intercepting requests made over IIOP by a CORBA object, converting them to the equivalent HTTP requests, and converting the result back into a suitable CORBA response over IIOP. The locator object, attached to H2I proxy, would be responsible for locating the appropriate CORBA service location based on the HTTP request URL. Since the HTTP client would be connected to the WWW only through the H2I proxy, the locator would have the additional responsibility of detecting when the request made by the client could not be serviced by any available CORBA server and was in fact intended for an HTTP server, and forwarding the HTTP request unmodified to the HTTP destination. This work proposed a specification for IDL mapping of HTTP requests to implement the protocol conversion component of the proxies, and suggested various configurations of where the H2I and I2H proxies should be located, including in the client and server computers respectively, and ultimately integrating the functionality into the web browser and HTTP server respectively.

Although the H2I and I2H proxies were reportedly implemented and source code is available for both, none of their publications detail the IDL/HTTP mapping, the implementation of the proxy mechanisms, nor present results for a working system. Furthermore, the work used IDL instead of DII, thus it is not clear how or if service discovery - a feature which seems necessary for a universal solution - would function, or if the proxies could only convert request for CORBA objects for which IDLs had been included in the proxies. Finally, since the project's official conclusion in 1999, there appears to have been no further activity or use of the produced software, and although source code for the H2I and I2H proxies is available, it is not clear if it is complete or functional. Finally, no cases were found in the literature of this approach being used for web-based control of robots.

3.3 Server-Side Remote Code Invocation

All of the technologies below fall under the category of Web Services and require an HTTP server with proper support for the particular technology. The essence of all these technologies is to allow an HTTP server to invoke an application in response to a web-browser request.

3.3.1 CGI

The Common Gateway Interface (CGI) is a standard protocol for interfacing a web server with applications that reside on the same host. It allows clients, typically web browsers, to execute applications on the web server. These applications format their output as an HTML document that is returned to the requesting client. CGI was the first method that enabled server-side code invocation for a web client. Despite its many drawbacks, it is still used widely today to generate dynamic content due in large part to its development and deployment simplicity. The executed applications can be developed in any language; compiled or interpreted. CGI processes have a large process creation overhead associated with every request. The CGI process first clones the web server process, then executes the program in the clone's address space [25, 26]. The entire process life-cycle and environment is managed by the CGI framework, including environment variables and re-direction of stdin/out/err. CGI uses the child process's output stream (stdout) to create the final HTML document. Variables from forms or other client input are passed to the process though environment variables, and nearly every programming language has libraries to facilitate accessing these variables. From the application developer's point of view, writing a CGI application is identical to writing a standalone application that writes to stdout.

Rivera et al. [27] used CGI to implement an XML-RPC-based scheme to enable internet based control of collaborative robots, though their client was not a web browser.

The main advantages of CGI are that it is extremely easy to develop and deploy, and allows the free choice of any programming/scripting language. It's overwhelming disadvantage is the extremely high process creation overhead costs, and lack of persistence across requests. This overhead is large when the application being run is short and when the request-rate is anything above moderate, and even larger when an interpreter must be run as in the case of scripting languages. CGI also facilitates the mixing of application logic and data presentation, which is generally bad practice.

To address this process creation overhead, FastCGI (discussed below) was created. In the case of CGI scripts, an alternative to CGI are server modules discussed in section 3.4.

3.3.2 FastCGI

FastCGI was developed specifically to address the high overhead cost of CGI requests. Instead of creating a new environment and instance of the application for every request, FastCGI takes a different approach: It separates the application process from the HTTP server process, and defines an API by which the application can communicate with the HTTP server. On startup the application opens a listening socket or pipe and waits for connections. On a request, the server connects to this socket and a simple protocol is executed. The protocol allows the process to send and receive data to/from the server. In addition, the protocol multiplexes the single connection between several possible requests. In addition to removing CGI's overhead cost, FastCGI presents additional advantages including the separation of application and HTTP server, which in turn improves stability as a crash of either process will not bring down the other. Libraries to implement the FastCGI API exist for nearly all popular languages, both compiled and interpreted.

There are currently no published works detailing the use of FastCGI as part of a web-based robot control architecture.

3.3.3 Servlets

The servlet specification was developed by Sun MicroSystems for the Java enterprise edition platform. Hence, the original API specification was for the Java language. The servlet API defines a container object that interacts with an HTTP server and a number of servlet objects which are managed by the container object. To invoke servlets, an HTTP server is configured such that URLs matching a particular pattern are forwarded to the container object. All HTTP method data is passed to the container object, including complete URL, HTTP method and any parameters. The container determines which servlet should handle the request based on the URL, invokes the servlet and forwards all request data obtained from the HTTP server as well as a output stream object. The individual servlets provide the application logic which generally depends on the passed URL and parameters. The servlet writes to the supplied output stream any data that must be sent back to client, such as a HTML document.

The servlet specification provides no application logic, only a means by which to invoke modules that provide the logic based on specific URLs. A container object may interact with an HTTP server and manage any number of servlets. The container object is persistent across requests and so does not incur the process creation overhead associated with CGI. Furthermore, the specification includes methods by which servlets may maintain data across requests. Each request to a particular servlet is handled in its own separate thread, which allows for handling of multiple simultaneous request from clients. Servlets are implemented as an extension of an HTTP server, or less commonly as a server module in a manner very similar to FastCGI.

Although the original specification was for the Java language, servlet's have been implemented - to varying degrees of completeness and compliance to the standard - in other languages including C++.

Servlets generally offer good performance as they do not incur the overheads present in either CGI or FastCGI. In a comparison of Java servlets, CGI and FastCGI, Apte et al. [25] found that when application logic was simple, Servlets performed better than CGI or FastCGI for an equivalent task[4], due to the reduced process and connection creation overhead. However, when the application logic was CPU intensive, FastCGI performed better than Java servlets. In addition, like any Java-based solution, servlets

[4]The CGI and FastCGI equivalents were implemented in C++

require the Java Virtual Machine runtime environment, and thus use more memory than an equivalent C++ solution.

Advantages of servlets include: the ability to use powerful languages like Java or C++ including all libraries available in the language, generally good performance, and depending on the implementation, handling of each incoming request in a separate thread. Few native implementations of the servlet specification in C++ exist, none are open-source, and thus many of the architectural benefits of servlets are lost to the Java language. Servlet's disadvantage is that they impede, and in fact prevent, the separation of logic and presentation since the servlet is responsible for both the application logic and the document generation. This problem is marked enough that several additional tools have been created around servlets in an attempt to remedy it.

There are currently no published works detailing the use of servlets as part of a web-based robot control architecture.

3.3.4 Application Server

The most common method of implementing servlets is to extend the capabilities of an HTTP server directly or to develop the HTTP server around the servlet specification. The result is that the servlet specification is bound to the few HTTP servers that implement it. These HTTP servers become the application server. Most open source application servers such as Apache Tomcat provide nothing more than the HTTP servers and servlet implementation; creating the application logic is entirely up to the developer. In contrast, most commercial application servers additionally provide a large array of application logic that is largely domain specific (such as financial, human resources and asset management, and database back-ends) and offer other features such as load balancing and authentication. Some application servers provide functionality for communicating with CORBA objects. Application servers are generally large, complex, costly software packages, which often require vendor support to deploy and maintain.

Application server's advantages over products that simply implement the servlet API is the additional application specific functionality, configuration utilities and a support contract, though these come at a large financial cost.

There are currently no published works detailing the use of application servers as part of a web-based robot control architecture.

3.4 Server-Side Scripting

Server-side scripting is a commonly used method to generate dynamic web content. Several scripting languages are currently widely used for dynamic content creation; a subset of the most common and widely deployed is introduced below[5]. Scripting languages are generally dynamically typed, interpreted languages that run under a virtual machine which also handles memory management. The dynamic and interpreted nature makes them much easier to develop with than compiled languages, but also decreases their performance. Many scripting languages offer a bytecode or native compiler, which can increase runtime performance. A common role for scripting languages is as a glue layer between different software components [28]. In a web services framework, scripting languages would be the intermediary or glue between the client and the web-service endpoint. Server-side scripting can be and often is used to drive application logic when the logic is simple or primarily involves database access. However if the application logic is complex, scripting languages will likely not provide adequate performance. Server-side scripting languages' forte is in text processing and dynamically generating HTML documents from the results of some other programs that run the application logic. This allows the separation of data, generated by the application logic, and presentation,the HTML document generated by the script. All of the languages below require an HTTP server, with proper support for the particular interpreter, to receive client requests and invoke the local interpreter and script.

The languages are evaluated based on the following criteria:

1. CORBA support
2. Ease of deployment
3. Web related modules/support such as HTTP/HTML processing
4. Socket-based IPC support
5. Performance
6. Object Oriented programming support

CORBA support is desirable as one possible solution to web-based robot control is for a server-side script to provide the application logic; that is for a server-side script to communicate with the CORBA services directly. Another solution could be for the server-side script to act as an intermediary between the HTTP server and a separate application providing the application logic. In this scenario socket-based IPC would be necessary. Scripts are often deployed as CGI applications, which incurs all the

[5]Microsoft's ASP is another popular scripting language. However, it does not run on Linux and hence is not considered here

performance penalties associated with CGI as mentioned in section 3.3. However, plugin modules for several HTTP servers are available for these languages which overcomes this problem by not only maintaining the virtual machine or interpreter in memory, but also retain application variables and state across invocations. Some scripting languages even allow the scripts to be mixed within the HTML document, further facilitating deployment.

3.4.1 PHP

The PHP language was designed specifically for web programming and generating dynamic web content, and as such has a wide range of utilities and features to support this. It has a syntax similar to C/C++, and with the latest version supports some object oriented features. It has native support in many HTTP servers and can thus be embedded directly into HTML documents, making PHP scripts extremely easy to deploy. The PHP script is executed, and thus the final document created, when the HTML document is requested by the client. It can be compiled to bytecode using the Zend engine. PHP is currently one of the most widely used web scripting languages, and boasts a large and active community and numerous extension modules including several for HTML and SOAP document manipulation, HTTP request handling and socket based IPC. PHP bindings for the MICO implementations of CORBA exist, however the status of the project is not clear.

Mookiah [11] used PHP as an intermediary component between the HTTP server and a Java application to enable web-based control of a robot. The PHP script connected to the Java application using sockets, forwarded the request from the client to the Java application, and formatted the results into an HTML document. Yang et al. used the same technique in [18] to control a robotic arm from a web-based client.

3.4.2 Ruby

Ruby is a general purpose scripting language that has recently gained popularity due to its Rails framework. It is a fully object oriented language and its main goal is ease of use for the developer. Consequently, it consumes more memory and performs poorer than the equivalent PHP, Python or Perl script. Ruby also has a large and active community and numerous modules including socket based IPC, and a few HTML and HTTP utilities. Ruby scripts are deployed as script files which are run via CGI or more commonly using a server module. In contrast to PHP where the script is embedded into the HTML document, Ruby scripts generate the HTML document as their output. Currently, there is no bytecode compiler for Ruby. Ruby CORBA bindings for the TAO implementation of CORBA have been developed by Remedy IT. Although Ruby does not yet have official OMG language mapping, Remedy IT submitted a draft specification to OMG in June of 2007 [29]. No accounts were found of Ruby being used as part of a web-based robot control architecture.

3.4.3 Perl

Perl is also a general purpose scripting language and is the oldest of the five presented here. It is primarily a procedural language, though the latest version supports some object oriented features. It's overwhelming strength is text processing. Perl also has a large and active community and boasts the largest number of add-on modules of the four languages, including several for HTTP and HTML manipulation and socket-based IPC. It is often criticized for its difficult-to-learn syntax which contains a large number of special characters and keywords, and makes it difficult to read. Perl compilers are available that can improve performance. Perl scripts are deployed in the same manner as Ruby scripts. Perl bindings exist for the Orbit and MICO implementations of CORBA. They are both mature and complete, however the MICO binding has no maintainer. No accounts were found of Perl being used as part of a web-based robot control architecture.

3.4.4 Python

Python is also a general purpose scripting language. It supports many programming paradigms well (procedural, object oriented, imperative) and other partially (functional). Like Ruby, it was designed with minimizing developer effort in mind, and like Ruby this translates to lower performance, though not to the same extent. Python has an extensive collection of libraries, including several for HTTP/HTML manipulation and socket-based IPC, and is used in many web applications. Python can be easily compiled to bytecode by the Python interpreter. Python scripts are deployed in the same manner as Ruby and Perl. CORBA binding exists for omniORB and Orbit CORBA implementations, though Orbit binding are only partial. No accounts were found of Python being used as part of a web-based robot control architecture.

3.4.5 xSP

Originally implemented in Java and called Java Server Pages (JSP), this technique for server-side dynamic content generation has been implemented in other languages. JSP was developed to address the main drawback of servlets; combination of logic and presentation. xSP works similarly to PHP where application code written in C/C++, Java or some xSP specific derivative is embedded in the HTML document. In contrast to PHP, which is parsed when the document is requested, with xSP the HTML document is compiled prior to deployment as a shared library, bytecode or the equivalent servlet, depending on the implementation. This results in much higher performance than the interpreted equivalent (i.e. PHP or ASP).

The benefit of this approach is that it provides a higher level of abstraction to servlets, which is attractive when servlets are part of a web services solution. No accounts were found of xSP being used as part of a web-based robot control architecture. Deploying xSP requires considerably more and complex configuration than the other alternatives.

3.4.6 Comparison

Table 1 summarizes the relevant properties of each scripting language. Because it can be embedded directly into the HTML document, PHP is the easiest to deploy. Its main drawback is its lack of CORBA support. Ruby's strength is its current and compatible CORBA support, while its disadvantages include having relatively few number of HTTP/HTML packages, compared to the other languages, and poor performance. Perl boasts the largest number of modules, of the four dynamic languages, however its syntax makes it unattractive. Python and Ruby are the most attractive languages syntactically as they offer full Object Oriented (OO) support. xSP could offer the best performance, especially when using C++SP, and can potentially access a large number of existing libraries, including TAO and Miro, though difficult configuration makes it unattractive. Ultimately the most appropriate scripting language for this task depends on the chosen software architecture.

Language	**OO**	**Web Modules**	**Deployment**	**Compilable**	**CORBA Bindings**
Perl	partial	y	cgi/mod_perl	y	Orbit
PHP	partial	y	embedded/cgi/mod_php	y	MICO[1]
Python	y	y	cgi/mod_python	y	omniORB
Ruby	y	limited	cgi/mod_php	n	TAO
xSP	y[2]	y[2]	shared library/servlet	y	y[2]

[1]currently unmaintained
[2]whatever libraries/features are available to specific language

Table 1: *Comparison of server-side scripting languages*

3.5 Other Methods

3.5.1 mod_corba

Mod_corba is an Apache version one module that exposes the Apache "module" API as a CORBA object allowing any CORBA service to be used as an Apache module. This approach would have each Miro and drdcMiro service registered as an Apache module. This project was last updated in 2001 and has had no activity since then. Additionally, no documentation is available describing any part of the project such as API, installation or configuration.

3.5.2 mod_cbroker

Mod_cbroker is a module for Apache that operates as an HTTP/CORBA gateway, converting HTTP requests to CORBA object requests [30, 31]. It uses a servlet-like architecture and API on the HTTP server end for passing data back to the client. Figure 2 shows the resulting architecture for this solution. This approach moves the responsibility of providing the web interface down to the individual CORBA service. It requires that all CORBA services that wish to make themselves available to web-based client implement two interfaces defined by the mod_cbroker module: HTTP::Servlet and HTTP::Handler. Requests made by client are received by the HTTP server and forwarded to the mod_cbroker module. The module parses the URL and searches for the specified service in the CORBA NamingService. If found, it calls the HTTP::Servlet object within that CORBA object, which authorizes the transaction, creates an instance of the correct HTTP::Handler object to serve the request based on the request information passed to the Servlet object, and returns that object to the mod_cbroker module. Mod_cbroker invokes the *handle* method on the returned Handler object, passing to it the requested information and an HTTP::Stream object in the same manner as servlets. The *handle* method services the request and creates a reply document using the HTTP::Stream object. The reply document is returned to mod_cbroker, which is in turn returned to the client.

The advantages of this approach are that it results in a very scalable solution: each new CORBA service would only need to implement the two interfaces to make itself accessible to WWW clients. Changes to any service would not affect other services. This solution should also provide good performance as the module is implemented in C and the number of components are kept to a minimum. This solution could be implemented one CORBA object at a time, which would allow its adequacy to be evaluated early. Maintenance could be difficult as integrating the additional interfaces with the upstream Miro package may be impossible, forcing the changes to be reapplied every time the Miro package is updated. Furthermore, only text-based media is supported via the stream object passed to the handler method, thus images, video and sound could not be sent directly to the client. Finally, this approach would not work with the Player/Gazebo [32] simulation package. This last point, despite its other advantages, renders this solution unfeasible because Player/Gazebo is currently the only system being used for simulation for DRDC research.

Although this project appears to be active as the documentation was last updated in 2007, the last source package was released in 2004. Additionally, the documentation for this package is difficult to understand.

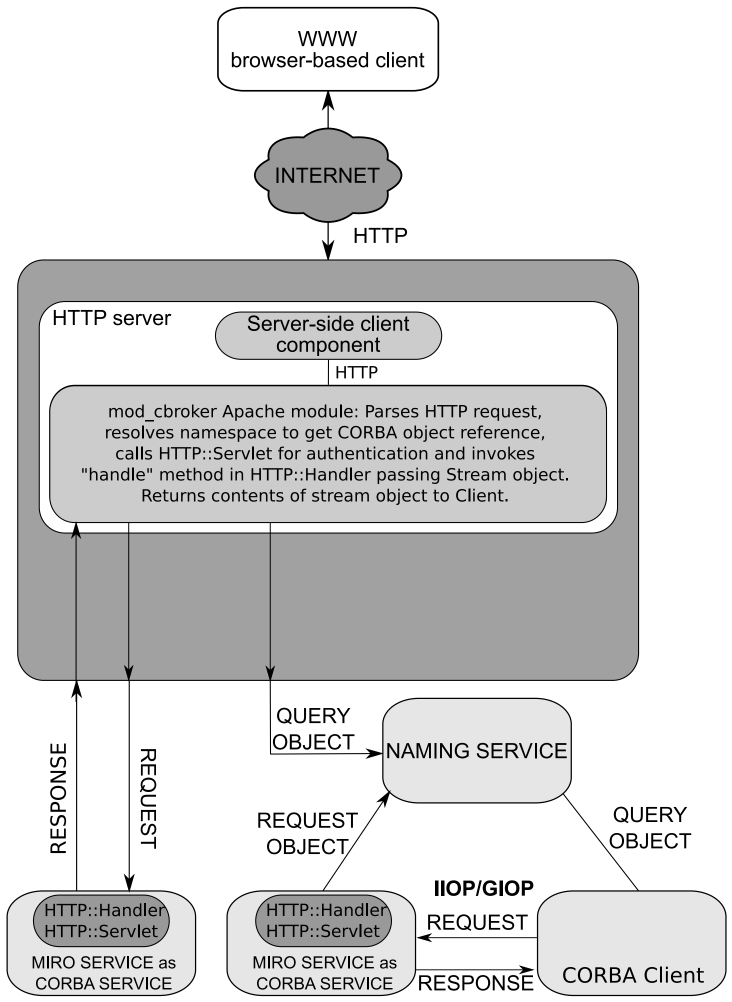
WWW
browser-based client
INTERNET
HTTP
HTTP server
Server-side client component
HTTP
mod_cbroker Apache module: Parses HTTP request, resolves namespace to get CORBA object reference, calls HTTP::Servlet for authentication and invokes "handle" method in HTTP::Handler passing Stream object. Returns contents of stream object to Client.
RESPONSE
REQUEST
QUERY OBJECT
NAMING SERVICE
REQUEST OBJECT
QUERY OBJECT
IIOP/GIOP
HTTP::Handler
HTTP::Servlet
MIRO SERVICE as CORBA SERVICE
HTTP::Handler
HTTP::Servlet
MIRO SERVICE as CORBA SERVICE
REQUEST
RESPONSE
CORBA Client

Figure 2: *Cbroker based solution.*

3.6 Web Services Models/Architectures

The concept of Web Service is relevant to this project as its aim is to expose a set of existing services to web-based clients, which is the function of web services. Web services can be implemented with any of the technologies described so far, and with a number of architectures. Some architectures require certain technologies, tools and additional components to implement, while others are little more than a set of guidelines. The architectures only impact the server-end of the system, and from the client's point of view there will be little difference other than the composition of the URL. This Section describes the three main web services architectures and examines how each could work with a CORBA-based system.

3.6.1 RESTfull Web Services

Representational State Transfer (REST) refers to a collection of principles that outline an architecture for client/server interaction, typically over HTTP[6]. Web services that apply or comply with these guidelines are said to be RESTfull These guidelines include [33]:

Resource-centric: Application data and functionality are divided into resources, and every resource is made addressable via a unique URL. Within the context of a robot composed of CORBA services, each offering a number of methods, this guideline could take the following shape.

```
http://www.robotServer.com/CORBAService1/method1/?par1=val1
http://www.robotServer.com/CORBAService1/method2/?par1=val1&par2=val2
http://www.robotServer.com/CORBAService2/method1/?par1=val1
http://www.robotServer.com/CORBAService2/method2/?par1=val1&par2=val2
```

This guideline has many supposed benefits including service discoverability.

Statelessness: A RESTfull Web Service specifies that interactions across requests be stateless. The server should retain no data about the client, including authorization, and all requests should include all necessary data for that transaction. This simplifies the server side implementation, but complicates the client.

Connectedness: In a RESTfull Web Service, resources provide connections (links) to other related resources. Often this includes links to all resources beneath the current resource in the resource tree; this provides a built in mechanism for resource service discovery. For example, the interface (i.e. web page) for the following resource

```
http://www.robotServer.com/CORBAService1/
```

[6]This is not a requirement of the RESTfull Web Services architecture, but for this project communication between client and server will be over HTTP

would provide links for all resources associated with it, namely

```
method1/
method2/
...
methodn/
```

Uniform Interface: In relation to HTTP, this means using the standard HTTP methods GET, HEAD PUT, POST and DELETE as they were intended, and designing your URL's and operations correctly. When this principle is adhered to, GET/HEAD requests become safe meaning they do not affect resources in any way and PUT/POST/DELETE requests are idempotent meaning calling any of these methods with the same arguments more that once is equivalent to calling it once. In the context of the above-mentioned URLs, if *methodn* only retrieved data, the GET method should be used. If it modifies the state of the server, the POST or PUT methods should be used.

Idempotent PUT/POST/DELETE operations are often difficult to guarantee depending on the service. Within the context of a robot for example, a command to rotate n degrees issued with either POST or PUT would not be idempotent: if the command were issued twice or more it would leave the robot in a different state than if it were issued once. One could instead specify a move command as an absolute location or coordinate, but this is not always feasible or even possible.

WADL. Web Application Description Language (WADL) is an XML file describing the services offered by a RESTfull web service. Clients can use a WADL for service discovery. However, because of the **Connectedness** principle, unlike WSDL described below WADL files are not necessary for service discovery or to interact with a RESTfull web service, and are thus seldom used.

This architecture is simple to implement as it does not rely on any specific technologies, and is in fact no more than a collection of guidelines to apply when designing how clients interact with a service. Its lack of tools to support implementation is the main critique against it.

3.6.2 WSDL-based Web Services

This type of Web Services architecture is currently popular, and includes a number of specific components and technologies; at its core are the Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP). This type of web service is also referred to as RPC, due to its underlying RPC-based architecture, or as SOAP-based, after its defining technology.

This type of Web Service uses HTTP only as a transport and container envelope, and places an additional SOAP envelope within the HTTP envelope. A SOAP envelope is an XML file that contains all the data required for the transaction including method name, parameters, authentication and possibly return types and faults. This manner of passing method name and parameters is a typical RPC architecture. In addition, the POST method is almost always used. A simple SOAP envelope is shown below:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    <soap:Body>
        <m:CORBAService1 xmlns:m="http://www.robotServer.com/">
            <method1>
                <arg1>value</arg1>
                <arg2>value</arg2>
            </method1>
        </m:CORBAService1>
    </soap:Body>
</soap:Envelope>
```

Through this envelope, we are invoking *method1* on *CORBAService1* with arguments *arg1* and *arg2*. From the client's point of view, the main visible difference compared to a RESTfull Web Service is that all transactions take place from a single URL. For the SOAP example shown above, the URL could be:

```
http://www.robotServer.com/
```

A WSDL file for this type of web service provides the means for service discovery, and is the WADL equivalent for SOAP-based services (except that a WSDL file is necessary whereas a WADL one is not). A WSDL file is an XML file that provides all the information necessary to interact with a service, including the service location (URL), all methods provided including parameter and return types, faults and more. WSDL is analogous to CORBA's IDL in that it describes all transactions and data types that can occur between a server and client.

Web browsers do not have the functionality to either parse WSDL files or create and parse SOAP envelopes directly, thus several additional steps and components are required for web browsers to access SOAP-based web services. First, a library to create, parse and marshal the request made by the browser into SOAP envelopes must be provided; this would be much like the stub and skeleton in IDL. Finally, this library must be made accessible to the browser. Several libraries currently exist that provide this functionality at different locations in the architecture and for different technologies including JavaScript, Java Servlets, CGI/FastCGI and various scripting languages.

A typical deployment, for example, would have the browser invoke a small application, via (Fast)CGI or a server script module, that would assemble the SOAP envelope and send it to the service running on the same computer. The service would parse the SOAP envelope, service the request, assemble the SOAP reply envelope and send it back to the SOAP client. Finally, the SOAP client would parse the SOAP envelope, create an HTML representation and return that to the web client. Although this requires more steps and is considerable more involved than the previous architecture, there are several tools that automate much of this.

One such tool worth mentioning due to its strong similarity with the CORBA application development model is gSOAP [34]. gSOAP is an open-source package that provides 3 components: a WSDL to C++ header file compiler, a C++ header file to C++ stub/skeleton compiler, and a client library. The stub and skeletons produced by the compiler serialize and deserialize the SOAP envelopes, and the client library provides among other things a transport layer between the SOAP client and server.

The claimed advantages of this architecture are its loose coupling of components and its quasi-statically-typed nature as a result of the WSDL. It's strength and prominent use scenario is in allowing different web services provided by different entities in the enterprise arena to interoperate by providing common interface standards in a language independent way. The main critiques against it are its complexity and loss of functionality of the "back" button and bookmarking ability for web browsers, though the latter two have workarounds.

3.6.3 Hybrid Web Services

Hybrid Web Services contain elements of the previous two architectures. Like the RESTfull services, they do not require additional tools to implement, and only rely on the HTTP protocol to operate. Like the SOAP-based services, they use an RPC-based architecture, and do not use HTTP in its intended manner. In a Hybrid Web Service, the URL for accessing a robot service could be:

```
http://www.robotServer.com/CORBAService/?method=methodname&par1=val1&par2=val2
```

or perhaps

```
http://www.robotServer.com/?CORBAService=servicename&method=methodname&par1=val1&par2=val2
```

All the transaction information is in the URL, as with RESTfull, however the method, and even the CORBA Service, to invoke the transaction is a parameter instead of a resource. Furthermore, any HTTP method (GET/POST/DELETE) could be used for the transaction. Hybrid web services can span a range of similarities between the SOAP and RESTfull architectures. There are no guidelines or tools to implement this architecture, and the developer can make their service as much like a RESTfull service or a SOAP-based service as they wish.

3.7 Toolkits

A vast number of toolkits and frameworks exist for any of the technologies mentioned above. This section will briefly introduce some of the more relevant ones for this application.

3.7.1 JavaScript/Ajax

These toolkits are aimed at facilitating the development of Ajax clients for web applications. They all provide similar resources, though some approach the development of Ajax web clients more like traditional GUI applications.

Google Web Toolkit
Google Web Toolkit (GWT) is a framework for developing rich web-based applications. It allows developers to do develop the application in Java, which is subsequently compiled to JavaScript by the provided compiler. It includes libraries for UI components, simple RPC mechanisms, browser compatibility and browser history management.

Wt
Wt is a toolkit for developing primarily event driven web-based GUIs in C++, in the same way traditional GUI applications are developed. In addition to UI widgets, it provides cross-browser compatibility and an event driven model. Similar to GWT, Wt web applications are written in one language and compiled into JavaScript and CGI client and server components respectively.

Dojo/Script.aculo.us/Yahoo Interface Library/EXT/Mootools
All of these toolkits provide JavaScript libraries for developing Ajax web clients and include a large array of UI elements that include animations and transition effects, drag and drop support, cross-browser compatibility, DOM manipulation, RPC, CSS, debugging and build control utilities.

JavaScript SOAP Client
This is a JavaScript object library to produce and consume SOAP messages. Instead of calling a SOAP client to perform the interaction with a SOAP server, this library allows the browser to communicate with the SOAP server directly. However, this precludes browsers that do not support JavaScript from accessing these services.

3.7.2 SOAP

gSOAP
gSOAP is a toolkit for developing SOAP client and servers in C++, and consists of a WSDL to C++ header file compiler, a C++ header file to C++ stub/skeleton compiler, and a client library. This library can be used to develop SOAP clients and servers from a WSDL definition of a service, with the generated code and client libraries handling all communication between client and server and serializing and deserializing of SOAP messages.

Scripting Languages
The Ruby, Python, PHP, and Perl scripting languages all provide a built-in SOAP library, and several have additional 3^{rd} party SOAP libraries.

Soap2Corba
Soap2Corba, or the SOAP to CORBA bridge parses incoming SOAP envelopes and from it constructs a DII CORBA call. The result of the CORBA call is assembled into a SOAP reply and sent back to the SOAP client. This project has been inactive since its last update in 2003, but it source code could prove useful in developing the server component.

4 Solutions

This section presents possible solutions to web-based control of Miro services. The service and client components are presented separately as they can be considered separate entities and largely rely on different technologies to implement. Despite this, the design of the client component impacts the design of the service provider component. Discussion of the relative advantages and disadvantages of each approach are left for Section 5.

4.1 Client Solutions

The client can take two forms: It can exist exclusively in the WWW browser, or it can contain an additional HTTP server-side component. The two components have distinct responsibilities, and the addition of the server-side client component does not affect the browser component's responsibilities. The two options are described bellow. Neither of the client components require CORBA support.

4.1.1 Browser Component

The browser component is responsible for initiating the HTTP request with the HTTP sever, and rendering the HTML response. A browser-only solution requires less components and technologies to implement and thus results in a slightly simpler overall architecture as the browser is a direct client of the service. However, because WWW browsers can only render HTML documents and static images [7], it forces the service endpoint to format its responses as HTML documents; this is inflexible, forces the mixture of logic and presentation in the service component and limits the type of clients it may service to WWW browsers. Additionally, web browsers cannot typically add envelops to HTTP requests such as SOAP or XML-RPC because they contain custom data, forcing the server to accept the most basic HTTP request in addition to any other format.

Implementation
In addition to HTML and CSS, the client browser component can use any of JavaScript, Ajax or Flash to create the user interface. Furthermore, excluding HTML these technologies could be used as little or as much as required to generate the desired interface, without impacting the service component.

4.1.2 Server-Side Component

A second alternative is to add a server-side component to the client[8] that would be located between the browser the service endpoint. Adding this component allows the service component to a) service only one request type, which can be designed or selected by the developer to allow additional envelopes to be attached to the request and b) provide its service in a representation that can more easily be consumed by clients other than a browser by providing replies in a format more universal than an HTML document. This component would be responsible for receiving the browser component's request, formatting it into an appropriate service request, making the request, producing an HTML representation from the result and finally sending the HTML document back to the browser component. This would be a lightweight component, and would allow the presentation to be changed independently of the service and vice-versa, so long as the interface between the service and server-side client component did not change. This component need not reside on the same HTTP server as the service endpoint. Finally, the existence of this component would be transparent to the browser, as in either case it would make an HTTP request and receive an HTML response.

Implementation
This component can be implemented using a script module, servlet, or (Fast)CGI. However, because this is a lightweight component, and because its main responsibilities are text manipulation and HTML document creation, a scripting language is the

[7]browsers can also render video and sound with the help of appropriate plugins

[8]"server" in this context corresponds to the HTTP server, not the service component of the solution.

most appropriate technology. Specifically, a scripting language such as PHP that was designed precisely for dynamic HTML content generation would be most appropriate.

Figure 3 shows the architecture of the possible client solutions. The server-side component formats and relays requests/responses between the browser component and service endpoint: it converts the HTTP request and parameters to whatever message format the service endpoint expects and formats the response to HTML. Without the server-side component the browser would communicate directly with the service endpoint. An additional level of flexibility with the second approach is that the messaging format the service endpoint expects could be changed without affecting the browser component. Finally, the server-side client component need not reside on the same computer as the service endpoint; indeed it could reside on any computer, which would slightly reduce the load on the robot resources though would likely introduce additional delay to requests. This would also facilitate a mechanism of service query: the server-side client components for all robots could reside on one computer. In addition to providing part of the client functionality, this computer could be responsible for querying all existing robots to determine if they are currently providing services. Client components could subsequently be disabled/enabled appropriately.

4.1.3 Provision

It should be noted that much of the functionality provided by the server-side client component could in fact be moved into the browser with the use of JavaScript. However, this would assume the browser supports JavaScript, which is not always the case. Additionally, the main point to be made here is not so much where the functionality provided by the server-side component resides, but its impact on the service provider and it's ability to provide an interface that can be consumed by a variety of clients. Additionally, in practice the two components will be somewhat bound if the browser client component is using technologies such as Ajax or Flash to modify and display the data. Although the browser component can be modified without affecting the server-side component, the reverse would not necessarily be true.

4.2 Service Endpoint Solutions

This component, which can also be referred to as the "Gateway", the "Application Server" or simply the "Service" will form the largest part of this project. It will be responsible for converting the request from the client component to the appropriate CORBA call to service the request, and returning the reply to the client in a manner that it can understand. Because this component will represent the bulk the effort required to service the request, its performance will be crucial in keeping the response time of a complete request low. The technologies chosen to implement this component must have CORBA support, and furthermore should take full advantage

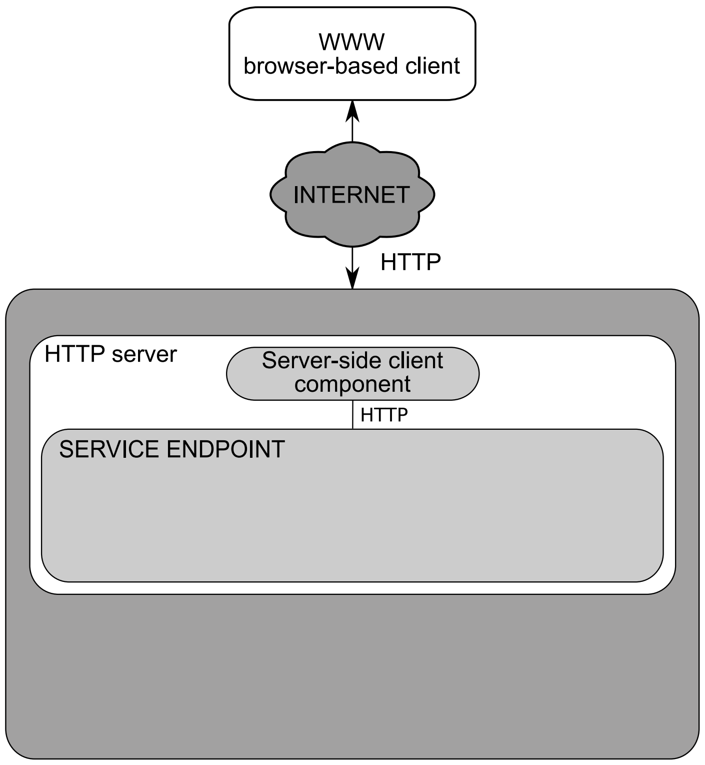

Figure 3: *Client solutions' architecture. The server-side component formats and relays requests/responses between the browser component and service endpoint. Without the server-side component the browser would communicate directly with the service endpoint.*

of the existing infrastructure as much as possible. To access the CORBA objects, this component can employ either IDL or DII. A message format between the client and service must be defined. The message format must include provisions to transmit all the data required to service the request, including HTTP request specific data such as request method, request parameters and client information, CORBA specific data such as the CORBA namespace, object or service name, method name and method parameter names and values, and response data including error messages. The choice of message format will be affected by the client architecture, the tools and technologies used to implement this component and the Web-Services architecture selected.

Potential Challenges
A potential point of conflict between CORBA and all the code invocation methods described below and all HTTP server-based code invocation methods should be highlighted. With all these techniques, including FastCGI, which is a separate process from the HTTP server, the HTTP server initiates the application to service the client's request, and thus effectively controls the servicing application's main execution loop, and in the case of FastCGI does so explicitly. Miro and drdcMiro clients that use the Publish-Subscribe or Publish-Server and Reactor design patterns also require that CORBA/Miro control the main execution loop. An interoperability challenge may arise out of the requirement for both technologies to control the main execution loop[9]. Two possible solutions are to avoid implementing clients with Subscribe and Reactor patterns and use polling clients as an alternative or to provide several threads of execution as required for each component.

IDL vs DII
Part of this component's role will be that of a Miro service client. A CORBA client can be implemented using either DII or IDL interfaces. DII would be preferable for a number of reasons. It would allow a single application to act as a universal CORBA client to all the services in the AISS suite, which would facilitate development and maintenance, and improve scalability. A single service endpoint application would reduce the amount of HTTP server configuration necessary and in turn simplify deployment. However, DII may present an additional challenge in acting as a client to services that implement the Publish pattern.

Alternatively, IDL, would require one service endpoint application for each IDL defined. This represents a higher development, deployment and maintenance burden than DII. Scalability would not be adversely affected by this approach as any number of application endpoints could be added to provide a WWW interface to new services. This approach could also present benefits for maintenance: modifications to one service endpoint application to fix bugs or accommodate changes in the corre-

[9]The mod_cbroker approach described in section 3.5, in contrast, would not suffer from this potential problem as the servicing routine would reside within the CORBA service.

sponding service would be isolated to one module, whereas with DII changes to the application to accommodate a new service or fix bugs could potentially impact the correct operation with other services. IDL does present two large advantages in that it would provide better performance than DII and would take greater advantage of current infrastructure and expertise as all AISS modules use IDL. The decision to use IDL or DII will not affect the selection of the technology used to implement this module.

The Service component can be implemented using the following technologies:

- Scripting module with CORBA support
- Servlet
- FastCGI

The details and implications of each are described below.

4.2.1 Script Module

As mentioned previously, the technologies and languages used to implement this component will have to support CORBA. Additionally, as mentioned in section 3.4 the Ruby scripting language has an up to date and active TAO bindings project lead by [29], and would be the most appropriate language to implement this approach. As with any scripting language solution, this solution would be best deployed as a server module such as Apache's mod_ruby. A scripting language such as Ruby is also more appropriate for manipulating text than a language like C++, which would be advantageous when parsing request and response messages from the client.

Technologies Involved
In addition to the Ruby TAO bindings, this solution would require an HTTP server that supports running the Ruby interpreter as a module.

Figure 4 depicts the architecture for this solution. The HTTP server would receive requests from the client and activate the script module to service the request. From the HTTP request data, the module would assemble the equivalent IDL or DII CORBA request and invoke the method on the required object. The result of the CORBA operation would be assembled into an appropriate response, HTML or otherwise, and sent back to the client. As mentioned previously, the server module that manages the script environment allows for data to be saved and used in subsequent requests allowing session management.

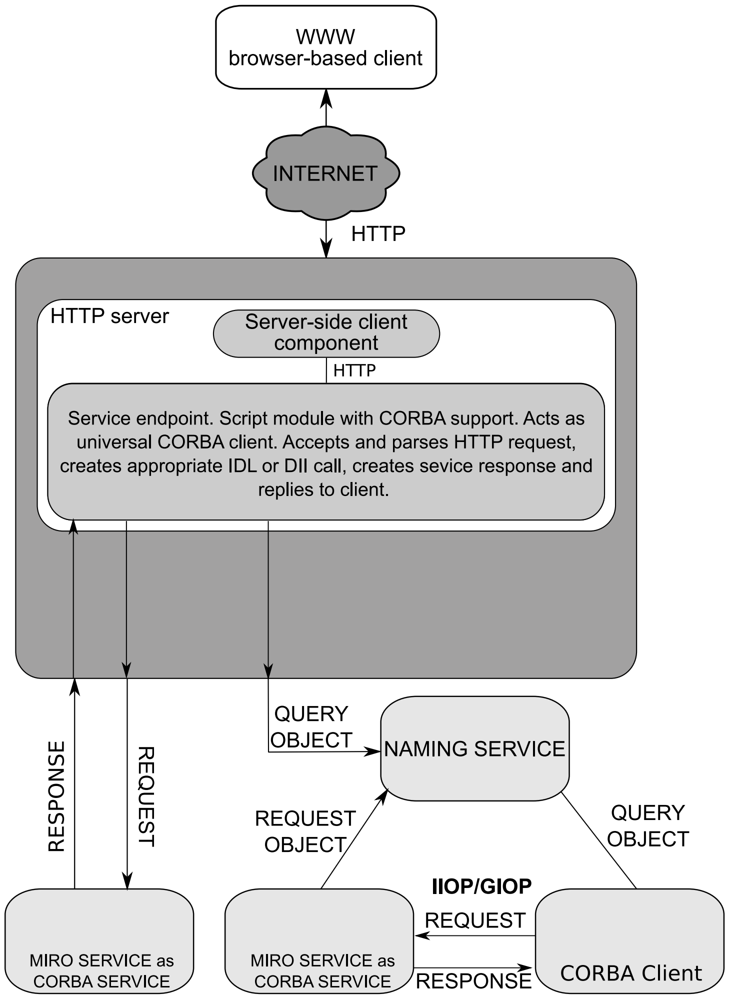
WWW
browser-based client
INTERNET
HTTP
HTTP server
Server-side client component
HTTP
Service endpoint. Script module with CORBA support. Acts as universal CORBA client. Accepts and parses HTTP request, creates appropriate IDL or DII call, creates sevice response and replies to client.
RESPONSE
REQUEST
QUERY OBJECT
NAMING SERVICE
REQUEST OBJECT
QUERY OBJECT
IIOP/GIOP
REQUEST
RESPONSE
MIRO SERVICE as CORBA SERVICE
MIRO SERVICE as CORBA SERVICE
CORBA Client

Figure 4: *Scripting module solution architecture.*

4.2.2 FastCGI

The second option for invoking server-side code from a web browser is the FastCGI protocol. The FastCGI specification is language independent and would thus allow this component to be developed in C++, taking full advantage of the existing AISS infrastructure and expertise. With the FastCGI specification, the servicing application runs as a separate process from the HTTP server, communicating through pipes or sockets. The end of a transaction usually involves closing the connection with the server, thus each new transaction requires a new connection be established; this is a potential performance bottleneck. As with any standalone application, parameters can be passed at runtime to alter its behaviour, which could have many advantages, especially during development. The FastCGI application must link to the FastCGI library, which handles all connections to the HTTP server. The FastCGI API is simple and library implementations exist in most languages including C++.

Technologies Involved
The technologies involved in this solution would be an HTTP server that supports FastCGI, the server module (such as mod_fcgid for Apache), the FastCGI client library, the C++ programing language and the full AISS stack including TAO and Miro.

Figure 5 depicts the architecture for this solution. The service endpoint would be started independently of the HTTP server and would listen for requests over pipes or TCP sockets from the FastCGI module within the server. The HTTP server would receive requests from the client and forward the request data to the FastCGI module. The FastCGI module would then open a connection to the service endpoint and forward the request information to the service endpoint. As with the previous solution, the service endpoint would activate the FastCGI script module to service the request. From the HTTP request data, the module assembles and invokes the appropriate CORBA request and prepares an appropriate reply. The reply would be sent back to the FastCGI module and the connection closed, indicating the end of the transaction. Finally, the FastCGI module would send the result of the transaction back to the client. The process persistence offered by FastCGI would, like the previous solution, allow data to persist across request thus providing a mechanism for session management.

4.2.3 Servlet/Application Server

The Servlet API is another method for server-side code invocation from a web client. Several implementations of the specification exist in C++ and Java, some of which are open source and others proprietary; however, none are open source native C++ implementations. Open-source C++ servlet implementations (such as CPPSERV) are implemented as server modules in a manner very similar to FastCGI, offering

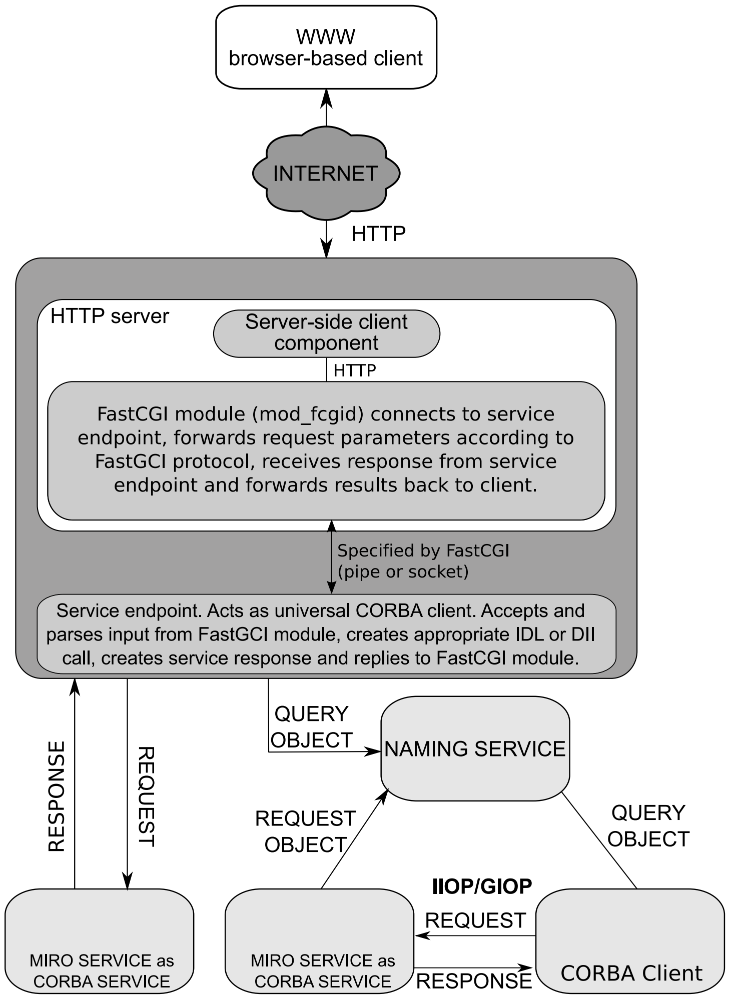
WWW
browser-based client
INTERNET
HTTP
HTTP server
Server-side client component
HTTP
FastCGI module (mod_fcgid) connects to service endpoint, forwards request parameters according to FastGCI protocol, receives response from service endpoint and forwards results back to client.
Specified by FastCGI (pipe or socket)
Service endpoint. Acts as universal CORBA client. Accepts and parses input from FastGCI module, creates appropriate IDL or DII call, creates service response and replies to FastCGI module.
RESPONSE
REQUEST
QUERY OBJECT
NAMING SERVICE
REQUEST OBJECT
QUERY OBJECT
IIOP/GIOP
REQUEST
RESPONSE
MIRO SERVICE as CORBA SERVICE
MIRO SERVICE as CORBA SERVICE
CORBA Client

Figure 5: *FastCGI solution.*

little performance advantage over FastCGI. A number of proprietary C++ application servers exist and although they would provide good performance, they would do so at a large development, configuration and financial expense. A C++ application server would allow this module to take full advantage of the existing infrastructure and expertise, but would tie the solution to a particular proprietary application server. An open source Java application server could be an alternative, though it could potentially perform poorly compared to a C++/FastCGI solution depending on the load characteristics, and would not take advantage of the AISS stack. Depending on the particular application server, the service module could be inside the HTTP server process, meaning it could not be started independently.

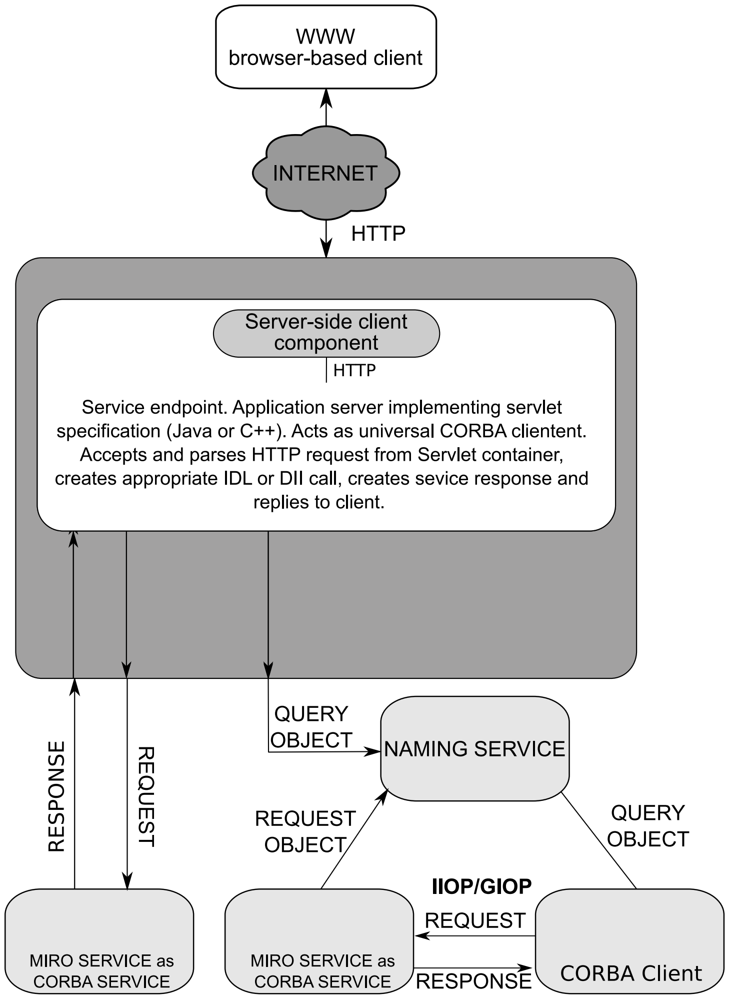
WWW
browser-based client
INTERNET
HTTP
Server-side client component
HTTP
Service endpoint. Application server implementing servlet specification (Java or C++). Acts as universal CORBA clientent. Accepts and parses HTTP request from Servlet container, creates appropriate IDL or DII call, creates sevice response and replies to client.
RESPONSE
REQUEST
QUERY OBJECT
NAMING SERVICE
REQUEST OBJECT
QUERY OBJECT
IIOP/GIOP
REQUEST
RESPONSE
MIRO SERVICE as CORBA SERVICE
MIRO SERVICE as CORBA SERVICE
CORBA Client

Figure 6: *Application server solution.*

Technologies Involved

The technologies involved in implementing this solution vary. If a native C++ application server is required, a product such as [35] could be used. Alternatively, Several open source native Java application servers exist based on the Apache HTTP server engine such as Apache's Tomcat and Geronimo, or RedHat's JBoss. Finally, an open source C++ servlet-like API could be provided by [36]. In addition to the application server, the full AISS stack could be used of if the application server is C++-based.

Figure 6 depicts the architecture for this solution. The HTTP server is extended to implement the servlet specification, and thus becomes an application server. This solution would operate in the same manner as the scripting module solution. As with the previous two solutions, application servers allow persistent storage of variables across requests allowing session management. The selected application server must also support the PHP interpreter if this solution is to be use in combination with the server-side client component.

5 Discussion

This section discusses the relative advantages and disadvantages of the solutions presented in the previous section with respect to the relevant criteria listed in section 2.3 and any additionally noteworthy attributes.

5.1 Client

The two choices for implementing the client are primarily architectural and will not affect the choice of technologies used to develop each of the components: The browser component will be implemented using XHTML, CSS, JavaScript and possibly Flash, and the server-side component would be implemented using PHP; the use of these technologies in their respective roles is well established. The discussion will then centre around the relative benefits and costs of the two architectures.

5.1.1 Browser-Only vs Server-Side Solution

As mentioned previously, a browser only solution forces the service endpoint to create and provide a presentation of the service and thus prevents separation of application logic and presentation. This impacts many of the criteria mentioned in section 2.3. In contrast, adding the server-side component (SSC) allows the service endpoint to interface with clients using a more compact, universal and ultimately appropriate messaging format (such as GData, JSON, etc.). This makes the following four points possible:

1. it allows the client UI to be developed and modified independently of the service endpoint,

2. it allows multiple web interfaces to be developed to satisfy different criteria (e.g. for mobile devices),
3. it facilitates implementation and maintenance of the service endpoint as it would no longer have to generate valid HTML documents, and
4. it allows any client that can understand the message format to consume this service. For example, one could develop a GTK/QT GUI client of a particular service(s) in C/Python/Ruby/Java that runs on a local computer, and without the requirement of the entire AISS stack.

Delay
Without the SSC, the browser communicates directly with the service endpoint. This solution does not incur the additional communication hop and associated delay that the addition of a SSC would. However, if the SSC and service endpoint reside on the same computer the delay would be minimal. Addition of the SSC also requires additional processing (and thus delay) to create and parse the message envelope.

Performance
As mentioned above, addition of a SSC would require additional processing and thus result in a small performance loss. This could be mitigated by placing SSC on different computer, though this would introduce more communication delay.

Deployment
The browser-only option involves less components and would thus require less effort to deploy. However, many HTTP servers support PHP natively and no additional configuration would be required making the additional deployment effort minimal.

Maintainability
The SSC solution would require a slightly higher maintenance effort as more components would be involved.

In summary, a browser only solution would be slightly simpler and yield slightly better performance at the expense of the four points mentioned above. The magnitude of the delay added by the SSC is currently unknown, thus its impact on the system and client experience as a whole is unknown. Furthermore, the benefits offered by the addition of the SSC component are known and significant.

5.2 Service

5.2.1 Script Module

Existing Infrastructure
Ruby's CORBA support is in the form of TAO bindings, which indirectly takes advantage of the underlying TAO infrastructure. This solution still does not take advantage

of Miro/drdcMiro packages or DRDC expertise with those packages. Developers could not rely on AISS personnel for expertise. Furthermore, it raises the entry barrier for current AISS personnel as they would have to learn both the Ruby language and its TAO bindings.

Multiple Clients
HTTP server modules such as Apache's mod_ruby provided a method for data persistence across requests. This could be used to implement a manner of session tracking to enforce one-client-at-a-time policy.

Delay
This solution does not require additional inter-process communication and thus minimizes communication delay. However, execution time for this solution would be substantially slower than the other options since Ruby is an interpreted and inherently poor performing language. This would be especially true if DII were used.

Performance
As with any interpreted language, Ruby's performance is considerably worse than compiled languages and even many scripting languages, including all the languages mentioned in section 3.4. Overall performance of this module would certainly be worse than the C++ equivalent. However it is not clear whether the performance would be adequate. Finally, a forthcoming Ruby bytecode compiler (YARV) should improve performance though it is not clear by how much, nor when the compiler will be mature enough for safe use.

Deployment
Deployment and configuration maintenance would be minimal for this solution and would only require enabling the Ruby module and creating a filter indicating which file types the module should service. Any HTTP server that supports running the Ruby interpreter as a module could be used.

Maintainability
Configuration related maintenance would be minimal as mentioned above. If IDL was selected, developing new components would require no more effort than a C++ based alternative as the Ruby CORBA package includes an IDL-to-Ruby compiler, and deploying them would be simpler than with any other solution. If the Dynamic Invocation Interface were used, changes to this module to accommodate new Miro and drdcMiro services would be minimal. However, developing a large project using dynamic languages generally requires more discipline and testing to compensate for errors discovered and fixed at compile time with compiled languages, thus any changes to this module should be followed by a series of tests to ensure proper operation.

Scalability
Using either IDL or DII would represent slightly less development and deploying effort

compared to the other solutions when adding new services due to Ruby's dynamic and interpreted nature. However, performance could become an issue if sufficient services are added and the request load becomes large enough.

Open Source
All the components required to implement this solution, including HTTP servers that support Ruby and the Ruby TAO bindings are distributed under open-source licenses.

Implementation
Implementation of this component using Ruby would represent the simplest solution due to its dynamic type system, automatic garbage collection, attractive syntax and easy deployment. Furthermore, the Ruby TAO bindings provided by [29] support direct inclusion of IDL in Ruby code, possibly further facilitating development.

Advantages
The main advantages of this solution are its ease of development, deployment and maintenance.

Disadvantages
Potentially inadequate performance - and certainly poorer than the alternatives - is the main disadvantage with this approach. It also does not take advantage of existing expertise, and thus raises the entry barrier for current AISS personnel. As mentioned, this solution could potentially require more maintenance effort and additional testing.

5.2.2 FastCGI

Existing Infrastructure
FastCGI allows the service endpoint to be developed in any language that supports pipes or sockets, and thus allows this solution to take full advantage of the existing AISS infrastructure and expertise. This, combined with FastCGI's relatively simple API, would present at a low entry barrier for current AISS personnel.

Multiple Clients
The service endpoint under FastCGI runs as a separate process that persists across requests and consequently so does its data. This can be used as a mechanism to implement session tracking and use policies.

Delay
Request servicing with FastCGI would be fast, as the servicing application would be developed with C++. However, the additional connection between the HTTP server and endpoint, particularly that fact that a new connection must be opened for each request, will be an additional source of delay.

Performance
Because this module can be implemented in C++ and because the FastCGI protocol imparts little additional overhead, performance in terms of CPU and memory usage will be good. As previously mentioned, the connection and communication between the servicing application and the HTTP server could potentially be a performance bottleneck.

Deployment
Deployment of FastCGI applications is slightly more complex than a script module; in addition to a FastCGI server module (such as Apache's mod_fastcgi or mod_fcgid), a modicum of HTTP server configuration is required. The ability to start and stop the servicing application independently of the HTTP server will also simplify deployment and development.

Maintainability
As with the scripting solution, either DII or IDL could be used to interface with CORBA objects. Selecting IDL would require the development of a new service endpoint and a small amount of additional configuration for each CORBA service. Similarly to the scripting module, the use of DII would result in few changes required to this module to accommodate new CORBA services. The FastCGI specification, server modules and client libraries are all mature and change little, and keeping up to date with the implementations should not present a maintenance burden.

Scalability
Because this module would be implemented in C++ and because the FastCGI overhead is small, this solution should scale well to increasing requests. If the request load increases to a point where one process cannot adequately service all requests, FastCGI offers two models for increasing capacity: Increasing the number of processes or adding multi-threading to one process. Both are viable methods of increasing the capacity of the application and would be investigated when necessary.

Open Source
All the components required to implement this solution, including HTTP servers that support FastCGI and the FastCGI module are distributed under open-source licenses.

Implementation
Implementing this component as a FastCGI module would be straight forward as the FastCGI API is relatively simple and several C/C++ libraries exist that abstract connections and communication with the HTTP server. Furthermore, the ability to (re)start the servicing application independently of the HTTP server would be advantageous during development. Additionally, the bulk of this application could first be developed as a standalone program, adding the FastCGI component at a later time.

Advantages
The advantages of FastCGI are many. The API is simple and several libraries exist that encapsulate all the implementations details. This solution allows us to take full advantage of the AISS existing infrastructure and expertise. Several HTTP servers support the FastCGI protocol so this solution would not be tied to a particular one. The separation of HTTP server and application processes facilitates development. It provides two options for increasing capacity. Finally, because the service endpoint and HTTP server communicate over sockets, they need not reside on the same computer. This would allow a large number of potential configurations. For example, one HTTP server could service requests for all robots and connect to the service endpoint located on the appropriate robot.

Disadvantages
FastCGI's main disadvantages are the potential communication bottleneck.

5.2.3 Servlet/Application Server

Existing Infrastructure
Whether or not this solution takes advantage of the existing AISS infrastructure will depend on the particular choice of application server. A C++ application server could allow this solution to utilize the full AISS stack and expertise, whereas a Java application server would not. Some commercial application servers, such as BEA's Tuxedo provide an implementation of CORBA which may or may not be fully compatible with TAO's implementation, and it might be considered redundant to use TAO's implementation instead of the included one. CORBA implementations provided by whatever Java package is used by the Java application server may similarly not be completely compatible with TAO.

Multiple Clients
Servlet containers allows data persistence across requests, and as with the previous two solutions, this could be used for session tracking to enforce one-client-at-a-time policy.

Delay
Because servlets run in the same process space as the HTTP server, no additional connections are required between the servlet container and the servlets that process the request, thus the additional communication overhead present with FastCGI would not be an issue. A Java application server would however introduce a JVM overhead associated performance loss and delay.

Performance
A native C++ application server would likely provide the best overall performance of all the solutions. For a Java application server, the JVM could become a bot-

tleneck [25] causing performance to drop below that of FastCGI. Finally, a C++ servlet-like API would provide similar performance to FastCGI.

Deployment
Deployment of application servers, whether C++ or Java based, commercial or open source is a complex affair requiring a large amount of relatively complex configuration in part because of the additional functionality they provide. Servlet-like server modules would require much less configuration, though more so than FastCGI.

Maintainability
Configuration related maintenance could require slightly more effort compared with other solutions if IDL were selected. As with the other solutions, a new service endpoint (i.e. a servlet) would have to be combined for each Miro service, which would require additional HTTP server configuration to deploy. As with the previous two solutions, using DII would require little or no change in the application or configuration to accommodate new CORBA services.

Scalability
Many application servers natively create a new thread for each servlet instance to service each request. This would represent the most scalable situation in that no additional developer effort would be required to ensure sufficient capacity.

Open Source
Several high quality native Java-based open-source application servers are available; the same is not true for C++. A handful of open-source C++ server modules are also available.

Implementation
This option would be the most complex to develop and deploy as the Servlet API is more complex than FastCGI's and could require substantial configuration. As with FastCGI, the service could first be developed as a standalone application and later integrated with the servlet infrastructure though it is less likely that this approach will be feasible. Finally, the servlet API varies slightly between implementations, which would likely tie this solution to the chosen application server.

Advantages
The main advantages of this approach are that it would potentially offer the best performance and scalability if a native C++ application server were used. Additionally, if the solution relied on C++, the full AISS stack could be used. Because native application servers are an extension of the HTTP server, they have access to many of its features such as error logging and thus result in a more integrated solution.

Disadvantages
Ironically, the main disadvantages associated with this application come about if one

attempts to take full advantage of its possible benefits. Although a commercial native C++ application server such as BEA's Tuxedo [35] would provide all the previously mentioned advantages, it would also represent the most complex and costly solution.

5.2.4 Uncertainties

It is not yet clear what the service load characteristics will be for this system: If the requests are relatively frequent but are serviced quickly, request servicing will be dominated by the request overhead (HTTP server processing and additionally connection creation/destruction for FastCGI) [25]. On the other hand, if requests are less frequent but require slightly longer to service, request servicing will be dominated by the service endpoint. For example, constructing and invoking the CORBA request and parsing the result into a return envelope. In the former case, servlets would likely provide better overall performance; in the latter a C++ based solution would; in which case FastCGI would be a better choice than an application server. However, the only assertion that can be made with some level of certainly is that under no condition would the script based solution perform better than either of the other two solutions. Even so, the overall servicing demand could be such that the Ruby-based solution provides adequate performance.

6 Conclusion

This section will outline the approach and technologies suggested to implement the solution to web based control, based on relative merits of the approaches presented in Section 5.

6.1 Technology Selection

6.1.1 Client

In light of the points raised in section 5.1.1, selecting a browser-only solution which poses an unknown performance benefit, over a solution that includes a server-side component which poses a known architectural benefit does not seem prudent. Consequently, a client solution that includes a server-side component should be adopted. The details for implementing each component are described below.

Browser Component
In addition to XHTML and CSS, JavaScript, Ajax and Flash can both be used to develop the browser UI component. JavaScript, Ajax and Flash have the same aim: to enable development of rich web clients. JavaScript is supported natively by browsers, whereas Flash support is enabled through a plugin or standalone Flash

player. JavaScript and Ajax are more open technologies than Flash; or at least better supported by open source implementations.

From a development and maintenance point of view it is easier to employ one technology instead of two for implementing a solution to a particular problem. Furthermore, the JavaScript development model is generally simpler than Flash due to the tools required to develop each; a text editor for the former, a proprietary IDE for the later. There is more tools support for developing Flash, provided by the IDEs, though this is more than made up for by the large number of available JavaScript toolkits and libraries. For these reasons, the browser UI should be developed primarily using JavaScript and Ajax. Flash does provide capabilities JavaScript and Ajax do not, such as animations, and if deemed necessary Flash can be added to the browser UI at a later time with minimal impact on the rest of the system,including existing UI components.

Server Side Component
The use of PHP to implement this module is a simple choice. PHP is designed specifically for generating dynamic HTML content. It's built-in functionality and deployment model are ideally suited for this application.

6.1.2 Service

A script based solution will yield the worst performance, regardless of the load characteristics. A C++ application server based solution could yield the highest performance, but at the expense of increased complexity and cost. FastCGI would yield slightly lower performance than an application server, but still vastly better than a scripting solution, and with much less development, deployment and maintenance characteristics than an application server. It follows that FastCGI is the best choice of technology for implementing the service endpoint component. The service endpoint should be developed in C++ taking full advantage of the AISS stack as necessary for CORBA communication and other functionality.

6.2 Implementation Plan

Development of the web-based robot control solution should take place stepwise with each step building upon the previous, and should proceed as follows:

- First, a simple Miro service and client should be selected as test case; the odometry or position control clients are candidates.
- A simple web-based client should be developed for the selected service using the IDL interface and deployed as CGI application (for simplicity) as a proof of concept and to highlight any possible compatibility issues between Miro/TAO/ACE

and the HTTP server. On each request this client would poll the Miro service and display the result as a simple HTML document.

- The client should then be augmented to use the FastCGI protocol, and deployed as such.
- The implementation of the service endpoints using the IDL versus the DII interface will have an effect on later decisions including the selection of a Web Services architecture, and thus should be selected next. Specifically, DII's additional implementation challenge, decreased performance and ability to interoperate with all Miro services should be compared against its development, maintenance, scalability and other benefits compared to IDL.
- Next, the interfaces of all Miro services should be examined to determine the requirement of a message format to adequately represent all the possible data types. Next, an appropriate message format should be selected. The choice of Web Services architecture will influence and be influenced by the message format, and thus should be decided concurrently.
- Once the message format has been selected, the service endpoint should be updated to respond to service requests using the message format.
- Next, the server-side client component should be developed to convert the service reply from the message format to an HTML document.
- The browser UI should be developed next, taking full advantage of JavaScript and Ajax. A number of the JavaScript/Ajax toolkits, mentioned in section 3.7, should be considered.
- A URL scheme to address all possible Miro services should be developed and implemented next, and will be influenced by the previously selected Web Services architecture.
- Once all components of the system are in place and working for one service, endpoints for each of the Miro services should be developed. If DII is selected, the universal client should be developed first, ensuring at can interface with all Miro services. The client UIs and server-side components for each Miro service should be developed subsequently. If IDL is selected, the service endpoint and client components should be developed concurrently.
- If desired or required, Flash components can be added to the client UIs to improve the interface at any point.
- A standalone client can optionally be developed as a sample of how to interact with the service from a non-browser client.

- Finally, developer and user documentation should be developed for all components.

6.3 Contingencies

The service endpoint will be the most important and largest individual component of entire project: it is the gateway between the WWW and CORBA, and essentially what makes web-based control of CORBA objects possible. Because the load characteristics of this system are not yet known, there exists the possibility that the request processing overhead mentioned in section 5.2.4 could dominate request servicing; the FastCGI communication overhead could then become a performance bottleneck prohibiting adequate performance. In this event, the service endpoint could be re-implemented as a C++ application server module and allow reuse of the majority of the developed code-base.

References

[1] Broten, G., Monckton, S., Giesbrecht, J., Verret, S., Collier, J., and Digney, B. (2004), Towards Distributed Intelligence - A high level definition, (DRDC Suffield TR 2004-287) Defence R&D Canada – Suffield.

[2] Various (2007), The ADAPTIVE Communication Environment (ACE). http://www.cs.wustl.edu/ schmidt/ACE.html.

[3] Various (2007), Real-time CORBA with TAO (The ACE ORB). http://www.cs.wustl.edu/ schmidt/TAO.html.

[4] Kumar, Atul, Gupta, Deepak, and Jalote, Pankaj (2002), Accessing CORBA Obejcts on the Web, In *Proceedings of the IADIS International Conference WWW/Internet, ICWI 2002*, pp. 485–490.

[5] Flanagan, David, Farley, Jim, Crawford, William, and Magnusson, Kris (1999), Java enterprise in a nutshell: a desktop quick reference, O'Reilly & Associates, Inc.

[6] Goldberg, K., Chen, B., Solomon, R., Bui, S., Farzin, B., Heitler, J., Poon, D., and Smith, G. (2000), Collaborative teleoperation via the Internet, In *Proceedings of the IEEE International Conference on Robotics andAutomation, ICRA'00.*, pp. 2019–2024.

[7] Schulz, D., Burgard, W., Cremers, A., Fox, D., and Thrun, S. (2000), Web interfaces for mobile robots in public places, *IEEE Robotics and Automation Magazine*, 7, 48–56.

[8] Edinbarough, Immanuel, Ramkumar, Manian, and Soundararajan, Karthik (2002), A web-based approach to automated inspection and quality controlof manufactured parts, In *Proccedings of the 2002 ASEE Annual Conference & Exposition: Vive L'ingenieur!*

[9] Ko, CC, Chen, B.M., Chen, J., Zhuang, Y., and Tan, K.C. (2001), Development of a web-based laboratory for control experiments on a coupled tank apparatus, In *IEEE Transactions on Education*, Vol. 44, pp. 76–86.

[10] Chatila, Wael (2006), AJAX Remote Controlled Lego Robot. http://waelchatila.com/2006/07/13/1152788433678.html.

[11] Mookiah, Prathaban (2006), Remote Robot Control with PHP. http://marc.info/?l=php-general&m=116232713608378&w=2.

[12] Google (2007), Google Web Toolkit - Build AJAX apps in the Java language. http://code.google.com/webtoolkit/.

[13] Deforche, Koen (2007), Wt: a C++ Web Toolkit. http://www.webtoolkit.eu/wt/.

[14] Yahoo (2007), Yahoo User Interface. http://developer.yahoo.com/yui/.

[15] Foundation, D. O. J. O. (2007), DOJO: The JavaScript Toolkit. http://dojotoolkit.org/.

[16] Various (2007), Script.aculo.us: Web 2.0 JavaScript. http://script.aculo.us/.

[17] Ext JS, LLC (2007), Ext JS - JavaScript Library. http://extjs.com/.

[18] Yang, S. H., Zuo, X., and Yang, L. (2004), Control system design for internet-enabled arm robots, In *Proceedings of the 17th international conference on Innovations in applied artificial intelligence*, pp. 663–672.

[19] Systems, Adobe (2007), Adobe Flex 2. http://www.adobe.com/products/flex/.

[20] Various (2007), Flash for Linux. http://f4l.sourceforge.net/.

[21] Various (2007), Flame project. http://www.flameproject.org/index.php/Main_Page.

[22] Various (2007), Red5 : Open Source Flash Server. http://osflash.org/red5.

[23] Various (1998), Official Record of the ANSA Project. http://www.ansa.co.uk/.

[24] Rees, O., Edwards, N., Madsen, M., Beasley, M., and McClenaghan, A. (1995), A Web of Distributed Objects, In *Proceedings of the Fourth International World Wide Web Conference*.

[25] Apte, V., Hansen, T., and Reeser, P. (2003), Performance Comparison of Dynamic Web Platforms, *Computer Communications*, 26, 888–898.

[26] Titchkosky, Lance, Arlitt, Martin, and Williamson, Carey (2003), A performance comparison of dynamic Web technologies, *ACM SIGMETRICS Performance Evaluation Review*, 31, 2–11.

[27] de Rivera, G. G., Ribalda, R., Colas, J., and Garrido, J. (2005), A generic software platform for controlling collaborative robotic system using XML-RPC, In *Proceedings of the International Conference on Advanced Intelligent Mechatronics, IEEE/ASME 2005*, pp. 1336–1341.

[28] Carlson, Kristofer J. (2005), Comparing Web Languages in Theory and Practice, Master's thesis, Bowie State University, Maryland, Europe.

[29] IT, Remedy, Remedy IT - Ruby CORBA Language Mapping. http://www.theaceorb.nl/en/rclm.html.

[30] GradSoft (2004), ModCBroker. http://www.gradsoft.com.ua/products/cbroker_eng.html.

[31] Schevchenko, R. and Doroshenko, A. (2002), A method of mediators for building web interfaces of CORBA distributed enterprise applications, In *Proceedings of the 2001 international conference on Information systems technology and its applications-Volume P-2*, pp. 53–63.

[32] Gerkey, Brian, Vaughan, Richard T., and Howard, Andrew (2003), The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323.

[33] Richardson, Leonard and Ruby, Sam (2007), RESTful Web Services, O'Reilly Media, Inc.

[34] Various (2007), gSOAP : SOAP C++ Web Services. http://www.cs.fsu.edu/ engelen/soap.html.

[35] Inc., BEA Systems (2007), Transaction Processing Monitor, BEA Tuxedo, TP Monitor, Monitor So-
lution, C, C++, COBOL, CORBA, C# applications to SOA, SOA Applications. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/tux.

[36] Knowledge, Total (2007), CPPSERV - Application server with Java Servlet-like API.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Suffield PO Box 4000, Station Main, Medicine Hat, AB, Canada T1A 8K6		2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED	
3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.) Internet Based Robot Control Using CORBA Based Communications			
4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.) Verret, S.; Collier, J.; Bertoldi, A.v.			
5. DATE OF PUBLICATION (Month and year of publication of document.) December 2009	6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.) 70		6b. NO. OF REFS (Total cited in document.) 36
7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.) Technical Memorandum			
8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.) Defence R&D Canada – Suffield PO Box 4000, Station Main, Medicine Hat, AB, Canada T1A 8K6			
9a. PROJECT NO. (The applicable research and development project number under which the document was written. Please specify whether project or grant.) 42zz78	9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)		
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC Suffield TM 2009-127	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)		
11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.) (X) Unlimited distribution ( ) Defence departments and defence contractors; further distribution only as approved ( ) Defence departments and Canadian defence contractors; further distribution only as approved ( ) Government departments and agencies; further distribution only as approved ( ) Defence departments; further distribution only as approved ( ) Other (please specify):			
12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.) Unlimited			

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

Researchers in the field of robotics have been seeking methods to both control and monitor their vehicles. Unfortunately the programs they have developed to perform these tasks are normally dependent on the robotic software infrastructure or are very difficult to understand for an outside user. This paper looks to tackle the problem of monitoring and controlling a robotics system using a web browser. The goal of this paper is to describe the potential for a system that will control and monitor a CORBA based robotics framework from a simple HTTP based browser.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

ACE
TAO
MIRO
CORBA
HTML
web based control
robotics